

NPS52-81-002

# NAVAL POSTGRADUATE SCHOOL

## Monterey, California



Z8000 Simulator

A Training Tool

John Moschovinos

April 1981

FEDDOCS  
D 208.14/2:NPS-52-81-002

Approved for public release; distribution unlimited.

Prepared for:  
Naval Postgraduate School  
Monterey, California

NAVAL POSTGRADUATE SCHOOL  
Monterey, California

Rear Admiral J. J. Ekelund  
Superintendent

D. A. Schradly  
Acting Provost

The work reported here was supported in part by the Department of  
Computer Science and the Naval Postgraduate School Computer Laboratory.

Reproduction of all or part of this report is authorized.

This report was prepared by:

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-81-002	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Z8000 Simulator A Training Tool		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John Moschovinos, Lieutenant Commander Hellenic Navy		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		12. REPORT DATE April 1981
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)  Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)  Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)  Simulator Z8000-Microprocessor Assembler		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)  This report explores the possibility of the use of a simulator as a training tool for the thorough learning of assembly language and the architecture of the microprocessors and the use of the simulator as a host computer for the building of an operating system in high level language. It also presents the implementation in a high level language of a simulator which simulates the CPU of an advanced microprocessor (Z8000) and his associated software that provides the debugging features and finally it presents the		

20. Abstract

implementation of one pass assembler supporting the assembly language of implemented simulator.



## TABLE OF CONTENTS

INTRODUCTION -----	5
APPENDIX A AM 28020 INSTRUCTION SET -----	16
APPENDIX B CHECK CONDITIONS -----	24
APPENDIX C PROGRAM LISTING -----	26
A. simulator.p ( including firstpart.i, secondpart.i thirdpart.i, fourthpart.i ) -----	26
B. LOAD.p -----	264
C. TRAP -----	264
D. VIRICALVERSION -----	297
APPENDIX D TEST PROGRAMS SOURCE LISTINGS -----	324
A. test3 -----	324
B. output3 -----	326
C. test2 -----	354
D. test9 (BUBBLE.L) -----	351
E. output9 -----	355
APPENDIX E SEQUENCES OF CALLS -----	362
APPENDIX G REQUIRED MODIFICATIONS FOR THE VM/370 SYSTEM -----	372
APPENDIX H USER'S INSTRUCTIONS -----	377
APPENDIX I 28020 ASM ASSEMBLER SPECIFICATIONS --	382
A. asm.p ( including assembler.i ) ---	387
B. initialize.p -----	388
C. table1 -----	389

F. table2 -----	544
E. table3 -----	546
G. test12 -----	552
H. test12.o ( output ) -----	554
I. test11 -----	558
J. test11.o ( output ) -----	562
INITIAL DISTRIBUTION LIST -----	565

## INTRODUCTION

### A. GENERAL

The Greek Navy has just recently decided to enter the computers era and for this reason has started to send Navy officers for the first time for the acquisition of the Computers science degree. The above officers will be the core group that will be used for two different tasks.

The first task is their involvement in the acquisition of new software and hardware and the second one is the transfer of the acquired knowledge to the non-commissioned officers that will support the new equipments.

The current trend in the Greek Navy is the procurement of powerful main frames that will fulfill only the operation requirements in peace and war time; therefore the utilization of the computer in peace period will be extremely low in the near future.

The other requirement, the training of the non - commissioned officers , will require the procurement of different types of micros in order the training follows the normal steps of the complexity of the architectures of micros to mini and then to vidi computers.

### B. PROBLEM

The acquisition of micros is cost effective if and only if the equipment is used for numerous students as it happens in NPS

and so different types of micros are supported or they are utilized for other purposes. Also the micros require extra maintenance that is costly especially if there are different types due to the high required salary of the specialized technicians and the cost to support the appropriate level of spare parts and special repair equipments.

As an alternative solution is the construction of a model in high level language that will simulate a microprocessor. The Model is an abstraction containing only the significant variables and relations and so it enables to concentrate only on the important facts and therefore the model stripped of the unnecessary details is easier to be understood.

The present report encountering the above problem presents the alternative solution and in order the model cover the micro and mini architectures it was decided to be simulated the Z8000 microprocessor which architecture is complex and more close to the architecture of the mini-computers.

#### COMPOSITION OF THE REPORT

The composition of the report is the following :

##### APPENDIX A :

Provides the listing of the AM Z8000 instruction set that is supported by the simulator.

##### APPENDIX B :

Provides a list of the conditions codes that are supported by the AM Z8000 instruction set.

#### APPENDIX C :

It lists the following programs

- simulator.p which simulates the Z8000 microprocessor in PASCAL language. The program is divided in five parts due to the size : firstpart.i, secondpart.i, thirdpart.i, fourthpart.i and simulator.p.
- LOAD.p which prepares the input file to the simulator in the case that the source file has been written in hexcode.
- TRAP which is composed from hex code and it is loaded automatically in the simulator memory. The hex code is used to serve the occurrence of a trap condition.
- VIRTUALVERSION which is a modification of the program simulator.p and supports virtual memory. The size of the virtual memory is defined by a constant declaration.

#### APPENDIX D :

It provides the sequence of calls of the procedures in the program simulator.p

#### APPENDIX E :



It lists three demonstration programs .

APPENDIX G :

It provides the required modifications in order the program simulator.p can be transferred in the VM/370 system.

APPENDIX H :

It provides the user's instructions.

APPENDIX I :

It describes the basic characteristics of the implemented assembler that supports the simulator and it presents the BNF grammar of the assembly language. Also it lists the following programs:

- assm.p : which is the implemented assembler in PASCAL language. The program is divided in two parts : assembler.i and assm.p.
- initialize.p : is a program that prepares the used by the assembler tables. This program must be executed only once with the installation of the assembler.
- table1, table2, table3 : they are the tables that are used by the assembler
- test10, test11 : they are two demo programs
- test10.o, test11.o : they are the output

listing of the above demo programs

APPENDIX A  
AM Z8000 INSTRUCTION SET

LOAD AND EXCHANGE

Mnemonics	Operands	Addr. Modes	Operation
CLR	dst	R	Clear
CLRB		IR	dst $\leftarrow$ 0
		DA	
		X	
EX	R,src	R	Exchange
EXB		IR	R $\leftarrow$ src
		DA	
		X	
LD	R,src	R	Load into Register
LDB		IM	R $\leftarrow$ src
LDL		IM	
		IR	
		DA	
		X	
		BA	
		BX	
LD	dst,R	IR	Load into Memory(Store
LDB		DA	dst $\leftarrow$ R
LDL		X	
		BA	
		BX	
LD	dst,IM	IR	Load immediate into
			Memory
LDB		DA	dst $\leftarrow$ IM
		X	

Mnemonics	Operands	Addr.Modes	Operation
LDA	R,src	DA X BA BX	Load Address R<source address
LDAR	R,src	RA	Load Address Relative R<source address
LDK	R,src	IM	Load Constant R<n (n = 0..15)
LDM	R,src,n	IR DA X	Load Multiple R<src (n consecutive words) (n = 1..16)
LDM	dst,R,n	IR DA X	Load Multiple(store Multiple) dst<R(n consecutive words) (n = 1..16)
LDR LDRB LDRL	R,src	RA	Load Relative R<src (range -32768...+32767)
LDR LDRB LDRL	dst,R	RA	Load Relative (store Relative) DST<R (range-32768...+32767)
POP PCPL	dst,R	R IR DA X	Pop dst<IR Autoincrement contents of R

Mnemonics	Operands	Addr.Modes	Operation
PUSH	IR,src	R	Push
PUSHL		IM	Autodecrement contents of R
		IR	IR<src
		DA	
		X	
-----			
ARITHMETIC			
-----			
ADC	R,src	R	Add
ADCB			R<R + src + carry
-----			
ADD	R,src	R	Add
ADDB		IM	R<R + src
ADDL		IR	
		DA	
		X	
-----			
CP	R,src	R	Compare with Register
CPB		IM	R - src
CPL		IR	
		DA	
		X	
-----			
CP	dst,IM	IR	Compare with Immediate
CPB		DA	dst - IM
		X	
-----			
DAB	dst	R	Decimal Adjust
-----			
DEC	dst,n	R	Decrement by n
DECB		IR	dst<dst - n
		DA	(n=1....16)
		X	
-----			
DIV	R,src	R	Divide (signed)
DIVL		IM	Word: Rn+1<Rn,n+1 ÷ src
		IR	Rn<remainder
		DA	Long Word: Rn+2,n+3
		X	< Rn..n+3 ÷ src
			Rn,n+1
			< remainder
-----			



Mnemonics	Operands	Addr.Modes	Operation
EXTS	dst	R	Extend Sign
EXTSB			Extend sign of low order
EXTSL			half of st through high
			order half of dst
INC	dst,n	R	Increment by n
INCB		IR	$dst \leftarrow dst + n$
		DA	(n = 1...16)
		X	
MULT	R,src	R	Multiply (signed)
MULTL		IM	Word: $Rn, n+1 \leftarrow Rn+1 \cdot src$
		IR	Long Word: $Rn..n+3$
		DA	$\leftarrow Rn+2, n+3 \cdot src$
		X	Plus seven cycles for each
			1 in the multiplicand
NEG	dst	R	Negate
NEGB		IR	$dst \leftarrow 0 - dst$
		DA	
		X	
SBC	R,src	R	Subtract with Carry
SBCB			$R \leftarrow R - src - carry$
SUB	R,src	R	Subtract
SUBB		IM	$R \leftarrow R - src$
SUBL		IR	
		DA	
		X	

# LOGICAL

Mnemonics	Operands	Addr. Modes	Operation
AND	R,src	R	AND
ANDB		IM	$R \leftarrow R \text{ AND } \text{src}$
		IR	
		DA	
		X	
CCM	dst	R	Complement
CCMB		IM	$\text{dst} \leftarrow \text{NOT } \text{dst}$
		IR	
		DA	
		X	
CR	R,src	R	OR
ORB		IM	$R \leftarrow R \text{ OR } \text{src}$
		IR	
		DA	
		X	
TEST	dst	R	TEST
TESTB		IR	$\text{dst OR } 0$
TESTL		DA	
		X	
TCC	cc, dst	R	Test Condition Code
TCCB			Set LSB if cc is true
XCR	R, src	R	Exclusive OR
XORB		IM	$R \leftarrow R \text{ XOR } \text{src}$
		IR	
		DA	
		X	

# PROGRAM CONTROL

Mnemonics	Operands	Addr.Modes	Operation
CALL	dst	IR	Call Subroutine
		DA	Autodecrement SP
		X	$\text{SP} \leftarrow \text{PC}$
			$\text{PC} \leftarrow \text{dst}$
CALLR	dst	RA	Call Relative
			Autodecrement SP
			$\text{SP} \leftarrow \text{PC}$
			$\text{PC} \leftarrow \text{PC} + \text{dst}$
			(range -4096 to +4096)
DJNZ DNJNZ	R, dst	RA	Decrement and Jump if Non-Zero
			$\text{R} \leftarrow \text{R} - 1$
			IF $\text{R} = 0$ : $\text{PC} \leftarrow \text{PC} + \text{dst}$
			(range -254 to 0)
IRET	--	--	Interrupt Return
			$\text{PS} \leftarrow \text{SP}$
			Autoincrement SP
JP	cc, dst	IR	Jump Conditional
		IR	if cc is true: $\text{PC} \leftarrow \text{dst}$
		DA	
		X	
JR	cc, dst	RA	Jump Conditional Relative
			if cc is true: $\text{PC} \leftarrow \text{PC} + \text{dst}$
			(range -256 to + 254)
RET	cc	-	Return Conditional
			if cc is true: $\text{PC} \leftarrow \text{SP}$
			Autodecrement SP
SC	src	IM	System Call
			Autodecrement SP
			$\text{SP} \leftarrow \text{old PS}$
			Push instruction
			$\text{PS} \leftarrow \text{System Call PS}$

# BIT MANIPULATION

Mnemonics	Operand	Addr.Modes	Operation
BIT	dst,b	R	Test Bit Static
BITB		IR	Z flag NOT dst bit specified by b
		DA	
		X	
BIT	dst,R	R	Test Bit Dynamic
BITB			Z flag NOT dst bit specified by contents of R
RES	dst,b	R	Reset Bit Static
RESB		IR	Reset dst bit specified by b
		DA	
		X	
RES	dst,R	R	Reset Bit Dynamic
RESB			Reset dst bit specified by contents of R
SET	dst,b	R	Set Bit Static
SETB		IR	Set dst bit specified by b
		DA	
		X	
SET	dst,R	R	Set Bit Dynamic
SETB			Set dst bit specified by contents of R
TSET	dst	R	Test and Set
TSETB		IR	S flag < MSB of dst
		DA	dst < all 1s
		X	

# ROTATE AND SHIFT

Mnemonics	Operand	Addr.Modes	Operation
RLDB	R,src	R	Rotate Digit Left
RRDB	R,src	R	Rotate Digit Right
RL	dst,n	R	Rotate Left
RLB		R	by n bits (n = 1,2)
RLC	dst,n	R	Rotate Left through Carry
RLCB		R	by n bits (n = 1,2)
RR	dst,n	R	Rotate Right
RRB		R	by n bits (n=1,2)
RRC	dst,n	R	Rotate Right through Carry
RRCB		R	by n bits (n = 1,2)
SDA	dst,R	R	Shift Dynamic Arithmetic
SDAB			Shift dst left or right by
SDAL			contents of R
SDL	dst,R	R	Shift Dynamic Logical
SDLB			Shift dst left or right by
SDLL			contents of R
SLA	dst,n	R	Shift Left Arithmetic
SLAB			by n bits
SLAL			
SLL	dst,n	R	Shift Left Logical
SLLB			by n bits
SLLL			
SRA	dst,n	R	Shift Right Arithmetic
SRAB			by n bits
SRAL			
SRL	dst,n	R	Shift Right Logical
SRLB			by n bits
SRLl			



# BLOCK TRANSFER AND STRING

Mnemonics	Operands	Addr. Modes	Operation
CPD	Rx,src	IR	Compare and Decrement
CPDB	Ry,cc		Rx-src Autodecrement src address $Ry < Ry - 1$
CPDR	Rx,src	IR	Compare,Decrement and Repea
CPDRB	Ry,cc		Rx - src Autodecrement src address $Ry < Ry-1$ Repeat until cc is true or
CPI	Rx,src	IR	Compare and Increment
CPIB	Ry,cc		Rx - src Autoincrement src address $Ry < Ry - 1$
CPIR	Rx,src	IR	Compare, Increment and Repe
CPIRB	Ry,cc		Rx - src Autoincrement src address $Ry < Ry - 1$ Repeat until cc is true or
CPSD	dst,src	IR	Compare String and Decremen
CPSDB	R,src		dst - src Autodecrement dst and src addresses $R < R - 1$
CPSDR	dst,src	IR	Compare String, Decr.and Re
CPSDRB	R,cc		dst - src Autodecrement dst and src addresses $R < R - 1$ Repeat until cc is true or

# BLOCK TRANSFER AND STRING MANIPULATION (Cont.)

Mnemonics	Operands	Addr.Modes	Operation
CPSI	dst,src	IR	Compare String and Increment
CPSIB	R,cc		dst - src Autoincrement dst and src addresses $R \leftarrow R - 1$
CPSIR	dst,src	IR	Compare String Incr. and Repeat
CPSIRB	R,cc		dst - src Autoincrement dst and src addresses $R \leftarrow R - 1$ Repeat until cc is true or $R = 0$
LDD	dst, src	IR	Load and Decrement
LDDB	R		dst src Autodecrement dst and src addresses $R \leftarrow R - 1$
LDDR	dst, src	IR	Load, Decrement and Repeat
LDDRB	R		dst < src Autodecrement dst and src addresses $R \leftarrow R - 1$ Repeat until $R = 0$
LDI	dst,src	IR	Load and Increment
LDIB	R		dst < src Autoincrement dst and src addresses $R \leftarrow R - 1$

# BLOCK TRANSFER AND STRING MANIPULATION (Cont.)

Mnemonics	Operands	Addr. Modes	Operation
LDIR LDIRB	dst,src R	IR	Load Increment and Repeat dst<src Autoincrement dst and src addresses R<R - 1 Repeat until R = 0
TRDB	dst,src	IR	Translate and Decrement dst<src (dst) Autoincrement dst address R<R - 1
TRDRB	dst,src R	IR	Translate, Decrement and Repeat dst<src (dst) Autodecrement dst addresses R<R - 1 Repeat until R = 0
TRIB	dst,src R	IR	Translate and Increment dst<src (dst) Autoincrement dst address R<R - 1
TRIRB	dst,src	IR	Translate, Increment and Repeat dst<src (dst) Autoincrement dst address R<R - 1 Repeat until R = 0
TRTDB	src 1 src 2, R	IR	Translate and Test, Decrement RH1<src 2 (src 1) Autodecrement src 1 address R<R - 1

# BLOCK TRANSFER AND STRING MANIPULATION (Cont.)

Mnemonics	Operands	Addr. Modes	Operation
TRTDRB	src 1, src 2, R	IR	Translate and Test Decrement and Repeat $RH1 \leftarrow \text{src 2 (src 1)}$ Autodecrement src 1 address $R \leftarrow R - 1$ Repeat until $R=0$ or $RH1=0$
TRTIB	src 1, src 2, R	IR	Translate and Test, Increment $RH1 \leftarrow \text{src 2 (src 1)}$ Autoincrement src 1 address $R \leftarrow R - 1$
TRTIRB	src 1, src 2, R	IR	Translate and Test, Increment and Repeat $RH1 \leftarrow \text{src 2 (src 1)}$ Autoincrement src 1 address $R \leftarrow R - 1$ Repeat until $R = 0$ or $RH1 = 0$
INPUT/OUTPUT			
IN	R, src	IR	Input
INB		DA	$R \leftarrow \text{src}$
IND	dst, src	IR	Input and Decrement
INDB	R		$\text{dst} \leftarrow \text{src}$ Autodecrement dst address $R \leftarrow R - 1$
INDR	dst, src	IR	Input, Decrement and Repeat
INDRB	R		$\text{dst} \leftarrow \text{src}$ Autodecrement dst address $R \leftarrow R - 1$ Repeat until $R = 0$

# INPUT/OUTPUT (Cont.)

Mnemonics	Operands	Addr.Modes	Operation
INI	dst,src	IR	Input and Increment
INIB	R		dst←src Autoincrement dst address $R \leftarrow R - 1$
INIR	dst,src	IR	Input, Increment and Repeat
INIRB	R		dst←src Autoincrement dst address $R \leftarrow R - 1$ Repeat until R = 0
OUT	dst, R	IR	Output
OUTB		DA	dst←R
OUTD	dst,src	IR	Output and Decrement
OUTDB	R		dst←src Autodecrement src address $R \leftarrow R - 1$
CTDR	dst,src	IR	Output, Decrement and Repeat
CTDRB	R		dst←src Autodecrement src address $R \leftarrow R - 1$ Repeat until R = 0
CUTI	dst,src	IR	Output and Increment
CUTIB	R		dst←src Autoincrement src address $R \leftarrow R - 1$ Repeat until R = 0
CPU CONTROL			
COMFLG	flags		Complement Flag (Any combination of C,L,S,P/



# CPU CONTROL

Mnemonics	Operands	Address Modes	Operation
DI	int		Disable Interrupt (Any combination of NVI,VI)
EI	int		Enable Interrupt (Any combination of NVI,VI)
HALT			HALT
LDCTL	CTLR src	R	Load into Control Register CTLR←src
LDCTL	dst CTLR	R	Load from Control Register dst←CTLR
LDCTLB	FLGR src	R	Load into Flag Byte Register FLGR←src
LDCTLB	dst FLGR	R	Load Program Flag Byte Register dst←FLGR
LDPS	src	IR DA X	Load Program Status PS←src
NOP			No Operation
RESEFG	flag		Reset Flag (Any combination of C,Z,S,P/V)
SETFLG	flag		Set flag (Any combination of C,Z,S,P/V)

## APPENDIX B

### CHECK CONDITIONS

The check conditions are the following:

CODE	MEANING	FLAG SETTINGS	BINARY
-----	-----	-----	-----
	Always false	--	0000
	Always true	--	1000
Z	Zero	Z=1	0110
NZ	Not zero	Z=0	1110
C	Carry	C=1	0111
NC	No carry	C=0	1111
PL	Plus	S=0	1101
MI	Minus	S=1	0101
NE	Not equal	Z=0	1110
EQ	Equal	Z=1	0110
OV	Overflow	V=1	0100
NOV	No overflow	V=0	1100
GE	Greater than or equal	(S XOR V) = 0	1001
LT	Less than	(S XOR V) = 1	0001
GT	Greater than	(Z OR (S XOR V)) = 0	1010
LE	Less than or equal	(Z OR (S XOR V)) = 1	0010
UGE	Unsigned	C = 0	1111

greater than

or equal

ULT	Unsigned	$C = 1$	0111
-----	----------	---------	------

less than

UGT	Unsigned	$((C=0) \text{ AND } (Z=0)) = 1$	1011
-----	----------	----------------------------------	------

greater than

ULE	Unsigned less	$(C \text{ OR } Z) = 1$	0011
-----	---------------	-------------------------	------

than or equal

# APPENDIX C PROGRAM LISTING

```

1
2
3
4
5
6
7
8  (*****
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 *
25 *
26 const
27
28     maxmem = 2500;(* available real memory *)
29     psarea = 1792;(* the starting address of the PSA area *)
30     systemstackpointer = 2304;(* default value for the system stack
31         pointer *)
32     maxinteg = 2147483647.0; (* maximum integer in the system *)
33     maxreg = 23;(* number of existed CPU registers in the simulator*)
34     (*R0..R15,RPC, IWO IFMPORARY REGISTERS , RFC, *)
35     (* REFRESH ,PSAPSEG, PSAOFF, and a copy of the R15*)
36     (* used in the interchange of system and normal mode *)
37     nsao1 = 22;(*PSAOFF register*)
38     nsapseq = 21;(*PSAPSEG register*)
39     refresh = 20;(*REFRESH register*)

```

## PROGRAM SIMULATOR.P

## CONSTANT DECLARATION

```

63 fcw = 19;(* cpu status word RFC and flags *)
64 temporary = 17;(*temporary registers R17,R18*)
65 counter = 16;(*counter register RPC*)
66 nspoff = 15;(*NSPOFF register*)
67 nspseg = 14;(*NSPSEG register*)
68 hexv = 16; (*base for hexadecimal numbers*)
69 sn = 14;(*system/normal FLAG*)
70 c = 7;(*acarry FLAG*)
71 z = 6;(*zero FLAG *)
72 s = 5;(*sign FLAG*)
73 pv = 4;(*parity/overflow FLAG*)
74 da = 3;(*decimal adjust FLAG*)
75 h = 2;(*half carry FLAG*)
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107

```

```
121
122
123
124
125 (*****
126
127
128
129 *****
130
131
132
133 type
134     alpha = packed array [0..1] of char;
135     bravo = array [0..maxmem] of alpha;
136     charlie = packed array [0..15] of boolean;
137     delta = array [0..maxreq] of charlie;
```

## TYPE DECLARATION

\*\*\*\*\*

---

( )

# VARIABLE DECLARATION

\*)

```

memory: bravo;(*real memory array for code and data*)
r : delta;(*cpu registers*)
ic: integer;(*decimal counter having the same value as the hex*)
    (*value of the RPC register*)
alldone : boolean;(*denotes the end of user's program execution*)
    (*due to normal termination, error condition *)
    (* or ABREAK PUNT *)
ldrogram : boolean;(*denotes that the user's program has been *)
    (*loaded*)
opcode: integer;(*decimal value of opcode*)
a, b, d: boolean;
(*temp variables used to substitute long terms in expressions*)
R11, EX1SB: boolean;
(*R11 in the case that R11 R,R opcode is executed restricts *)
(*the returned subroutine so as to return the value that exists*)
(*in the last four bits of the registers*)
(*EX1SB : extends the attribute of byte registers to all the *)
(*registers in the case of execution of the EX1SB opcode *)

ADDB, SUBB : boolean ;(*denotes that byte operation has been *)
    (*executed*)
ABREAKPOINT ,nhtime : integer;(*ABREAKPOINT denotes that the *)
(*user has specified a breakpoint in the execution of the user's*)
(*program.The program returns to the monitor if the breakpoint *)
(*address is encountered the nhtime
TRACE ,finished : boolean;(*TRACE denotes that the user has *)

```



```

(*specified the trace function for each instruction *)
(*finished : is used to denote that the program must terminate*)
next : integer;(*denote how many instructions will be traced*)
cc: boolean;(* test condition*)
-system : boolean ; (* denotes the mode of the operation system
                    or normal *)
ch: char;(*is used to read and write char to the input and *)
      (* output files and also to the screen*)
src, dst: integer;(*src and dst registers in the opcode *)
validch : set of char; (*set of valid characters in the simu-*)
      (*lator 'A'..'F','0'..'9'*)

```

3047  
3048  
3049  
3050  
3051  
3052  
3053  
3054  
3055  
3056  
3057  
3058  
3059  
3060  
3061  
3062  
3063  
3064  
3065  
3066  
3067  
3068  
3069  
3070  
3071  
3072  
3073  
3074  
3075  
3076  
3077  
3078  
3079  
3080  
3081  
3082  
3083  
3084  
3085  
3086  
3087  
3088  
3089  
3090  
3091  
3092  
3093  
3094  
3095  
3096  
3097  
3098  
3099  
3100  
3101  
3102  
3103  
3104  
3105  
3106  
3107  
3108  
3109  
3110  
3111  
3112  
3113  
3114  
3115  
3116  
3117  
3118  
3119  
3120  
3121  
3122  
3123  
3124  
3125  
3126  
3127  
3128  
3129  
3130  
3131  
3132  
3133  
3134  
3135  
3136  
3137  
3138  
3139  
3140  
3141  
3142  
3143  
3144  
3145  
3146  
3147  
3148  
3149  
3150  
3151  
3152  
3153  
3154  
3155  
3156  
3157  
3158  
3159  
3160  
3161  
3162  
3163  
3164  
3165  
3166  
3167  
3168  
3169  
3170  
3171  
3172  
3173  
3174  
3175  
3176  
3177  
3178  
3179  
3180  
3181  
3182  
3183  
3184  
3185  
3186  
3187  
3188  
3189  
3190  
3191  
3192  
3193  
3194  
3195  
3196  
3197  
3198  
3199  
3200  
3201  
3202  
3203  
3204  
3205  
3206  
3207  
3208  
3209  
3210  
3211  
3212  
3213  
3214  
3215  
3216  
3217  
3218  
3219  
3220  
3221  
3222  
3223  
3224  
3225  
3226  
3227  
3228  
3229  
3230  
3231  
3232  
3233  
3234  
3235  
3236  
3237  
3238  
3239  
3240  
3241  
3242  
3243  
3244  
3245  
3246  
3247  
3248  
3249  
3250  
3251  
3252  
3253  
3254  
3255  
3256  
3257  
3258  
3259  
3260  
3261  
3262  
3263  
3264  
3265  
3266  
3267  
3268  
3269  
3270  
3271  
3272  
3273  
3274  
3275  
3276  
3277  
3278  
3279  
3280  
3281  
3282  
3283  
3284  
3285  
3286  
3287  
3288  
3289  
3290  
3291  
3292  
3293  
3294  
3295  
3296  
3297  
3298  
3299  
3300  
3301  
3302  
3303  
3304  
3305  
3306  
3307  
3308  
3309  
3310  
3311  
3312  
3313  
3314  
3315  
3316  
3317  
3318  
3319  
3320  
3321  
3322  
3323  
3324  
3325  
3326  
3327  
3328  
3329  
3330  
3331  
3332  
3333  
3334  
3335  
3336  
3337  
3338  
3339  
3340  
3341  
3342  
3343  
3344  
3345  
3346  
3347  
3348  
3349  
3350  
3351  
3352  
3353  
3354  
3355  
3356  
3357  
3358  
3359  
3360  
3361  
3362  
3363  
3364  
3365  
3366  
3367  
3368  
3369  
3370  
3371  
3372  
3373  
3374  
3375  
3376  
3377  
3378  
3379  
3380  
3381  
3382  
3383  
3384  
3385  
3386  
3387  
3388  
3389  
3390  
3391  
3392  
3393  
3394  
3395  
3396  
3397  
3398  
3399  
3400  
3401  
3402  
3403  
3404  
3405  
3406  
3407  
3408  
3409  
3410  
3411  
3412  
3413  
3414  
3415  
3416  
3417  
3418  
3419  
3420  
3421  
3422  
3423  
3424  
3425  
3426  
3427  
3428  
3429  
3430  
3431  
3432  
3433  
3434  
3435  
3436  
3437  
3438  
3439  
3440  
3441  
3442  
3443  
3444  
3445  
3446  
3447  
3448  
3449  
3450  
3451  
3452  
3453  
3454  
3455  
3456  
3457  
3458  
3459  
3460  
3461  
3462  
3463  
3464  
3465  
3466  
3467  
3468  
3469  
3470  
3471  
3472  
3473  
3474  
3475  
3476  
3477  
3478  
3479  
3480  
3481  
3482  
3483  
3484  
3485  
3486  
3487  
3488  
3489  
3490  
3491  
3492  
3493  
3494  
3495  
3496  
3497  
3498  
3499  
3500

PROCEDURE error

# FUNCTION

----- It prints an error message to the user and it suspends the execution of the user's program. In the case of error number (7) it skips the instruction and continues the execution. Also for debugging purpose it prints the contents of the program counter where the error has been occurred.

## PARAMETERS

```
-----by value: 'n' defines the number of error con-
dition that has occurred. The range of the parameter
is 1..12.
```

LOCAL VARIABLES

	None.
.....	.....

GLOBAL VARIABLES

```
-----: - all done suspends the user's program execution.
          Range 1 or F.
- ic contains the current value of the RPC
  register. Range 0..maxmem.
```

The procedure calls : No other procedures.

The procedure is called by : setindex, setmem, map, TRAP, TRFI, INPUT, INPUT, CALLREL, EXECUTE.

100

```
procedure error(n: integer);
begin(*
alldone := false;
```

340  
341  
342  
343  
344  
345  
346  
347

```

361 write(' ** ');
362 case n of
363 1:
364 write('PC G1 or L1 available memory');
365 2:
366 write(' Stack OV or UDF');
367 3:
368 write('Mem. viol..Refer. to location LT 0');
369 4:
370 write('Req. num. GT 23 or LT 0');
371 5:
372 write('Mem. viol..Refer. to location GT',
373 ' max. memory');
374 6:
375 write('Istr. not implem. in the NSM');
376 7:
377 begin(*3*)
378 write('The istr. MBIT,MREQ,MRES,MSET, are',
379 ' not impl. ');
380 alldone := true;
381 ic := ic + 1
382 end;(*3*)
383 begin(*4*)
384 write('Invalid instr. ');
385 alldone := true;
386 end;(*4*)
387 begin(*5*)
388 write('Odd addr. encountered ');
389 alldone := true;
390 end;(*5*)
391 begin(*6*)
392 write('Odd numb. for double req. ');
393 alldone := true;
394 end;(*6*)
395 writeln(' ** PC = ', ic: 4);
396 alldone := not alldone
397 (* changes the value to true if not error condition 7 or 8 *)

```

```
(* has occurred *)  
end;(*! error*)
```

424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467

(\*\*\*\*\*)

PROCEDURE uperlow

FUNCTION

----- Specifies the MSB and LSB of any register.  
The system supports 4 kinds of registers and  
the correspondence of type of the register and the  
value of the variable length is the following :  
length = 0 one byte register.  
length = 1 one word register.  
length = 2 double word register.  
length = 4 two double words register.  
The registers R0..R7 consist of two bytes registers that  
can be addressed separately. The correspondence between  
the number of the register and the byte register is the  
following :

length = 0	
Register No	byte register
0	RH0
1	RH1
2	RH2
3	RH3
4	RH4
5	RH5
6	RH6
7	RH7
8	RL0
9	RL1
10	RL2
11	RL3
12	RL4
13	RL5
14	RL6
15	RL7

Therefore if the length = 1 or 2 or 4 then the MSB is equal to 15 and the LSB equal to 0.

If the length = 0 then depending on the type of the byte register we have :

MSB	LSB
RH	15
RL	7
	0

In the case of the execution of the EXTSB opcode it is extended the low byte register to all the registers.

#### PARAMETERS

----- by value:

- index denotes the number of the register, the range is 0 .. 23.
- length denotes the length of the register, the range is 0..4.

by reference:

- upper denotes the MSB, the range is 7 or 15.
- low denotes the LSB, the range is 0 or 8.

#### GLOBAL VARIABLES

----- : - EXTSB denotes that EXTSB opcode is executed.  
the value is used but not changed. Range I or F.

#### LOCAL VARIABLES

----- : - The parameters upper and low.

The procedure calls : No other procedures.

The procedure is called by : setreg, retvalreq, onetwocompl, OKANDXORAND, shiftrightleft, BITMAN, ARITHMETIC, MULTIPLIER.

\*\*\*\*\*)

procedure upperlow(index, length: integer; var upper,



```

        low : integer);

begin(*1*)
  if lenath = 0 then begin(*2*)
    if index > 7 then begin(*3*)(*means RL register*)
      upper := 7;
      low := 0
    end else begin(*3,4*)(*means RH register*)
      upper := 15;
      low := 8
    end(*4*)
  end else begin(*2,5*)(*single word register*)
    upper := 15;
    low := 0
  end(*5*)
end(*1*)
if EXTSB then begin(*6*)(*in the case of the extension of sign*)
  upper := 7>(*are used the low registers*)
  low := 0
end(*6*)
end(*upperlow 1*)

```

```

664  (*****
665  *****
666  *****
667  *****
668  *****
669  *****
670  *****
671  *****
672  *****
673  *****
674  *****
675  *****
676  *****
677  *****
678  *****
679  *****
680  *****
681  *****
682  *****
683  *****
684  *****
685  *****
686  *****
687  *****
688  *****
689  *****
690  *****
691  *****
692  *****
693  *****
694  *****
695  *****
696  *****
697  *****
698  *****
699  *****
700  *****
701  *****
702  *****
703  *****
704  *****
705  *****
706  *****
707  *****

        PROCEDURE setindex

FUNCTION
----- : The procedure has double function.It checks if the used
          number of the register is correct and transforms the
          symbolic number of a byte register to the correct number.
          Error condition occurs if it is encountered:
            - odd number for double register.
            - number of register greater than 23 or less than 0.

PARAMETERS
----- by value : None
          by reference : - length defines the length of the register.
                        Range is 0..4.
                        - index defines the number of the register.
                        Range is 0..23.

GLOBAL VARIABLES
----- : None.

LOCAL VARIABLES
-----: - The parameters length, index.

The procedure calls : The procedure error.
The procedure is called by : setreq,retvalreq,onetwocompl,
GRANDXOR,ADD,shiftrightleft,BITMAN,ARITHMETIC,MULTIPLIER.
*****
        procedure setindex(var length, index: integer);
        begin(**)
          if (length = 2) and (odd (index)) and (index <> 17) then
            error(10);(* odd number is used for double word register*)

```

```

721
722
723
724
725   if length = 0 then begin(*2*)
726     if (index > 7) and (index < 16) then(*index for registers
727       RL or RH *)
728       index := index - 8;(*index of registers RL1..7 = 8..15*)
729       length := 1
730       (*index of registers RH 0..6 = 0..7*)
731   end;(*2*)
732   if (index > 25) or (index < 0) then begin (*3*)
733     (*range of the registers*)
734     index := 0;
735     error(4);
736     end;(*3*)
737   end;(*1 setindex*)
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761

```

```

784 (*****
785 *****
786 *****
787 *****
788 *****
789 *****
790 *****
791 *****
792 *****
793 *****
794 *****
795 *****
796 *****
797 *****
798 *****
799 *****
800 *****
801 *****
802 *****
803 *****
804 *****
805 *****
806 *****
807 *****
808 *****
809 *****
810 *****
811 *****
812 *****
813 *****
814 *****
815 *****
816 *****
817 *****
818 *****
819 *****
820 *****
821 *****
822 *****
823 *****
824 *****
825 *****
826 *****
827 *****

```

PROCEDURE setreg.

FUNCTION

-----: The procedure sets a register to a decimal value, trans-  
forming the decimal value to a binary one. Because the  
expected values may be greater than the maximum permitted  
integer value in the UNIX system, the procedure uses  
real numbers and it uses her mod operation. In the case  
of double registers the procedure first sets the low half  
part and then continues to the upper part. In the case of  
MULTI when the simulator has to set quadrable register  
the procedure is called twice in order to avoid over-  
flow in the real numbers of the UNIX system and first  
it sets the low double register and then the upper high  
double register.

PARAMETERS

-----: by value - length denotes the length of the register.  
Range is 0..23.  
- value denotes the value that the register  
will be set. Range is 0 ..2 to the power  
of 31.  
- index denotes the number of the register to  
be set. Range is 0..23.

by reference - None

GLOBAL VARIABLES

-----: 'r' represents the used register.

LOCAL VARIABLES

-----: - val is a copy of the parameter value. Range is  
0..2 to the power of 31.  
- ind is a copy of the parameter index. The range is

```

841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857

```

0..23.

- k is used as a subscript for the bits of the register.Range is 0..15.
- j used as an index for the for loop.Range is 1..2.
- l is a copy of the parameter length.Range is 0..2.
- upper,low denote the MSB and the LSB of the used register.

The procedure uses local variables for the parameters because the called procedures upperlow and setindex are called by reference and the procedure itself changes the value of the index of the register.

```

858 The procedure calls : the procedures upperlow and setindex.
859 The procedure is called by : IPAP , LPDS,KOIAIFSHIF1,compare
860 BLOCKTRANSFER,TRANSLATE,DAR,LDCIL,IRFI,common,inputreq,
861 INPUTOUTPUT,BITMAN,pushstack,CALLREI,DIVMULT,
862 ARITHMETIC,LOGICAL,compare,MULTIUPER,EX,LOAD,STORE,
863 boot,EXECUTE,JUMP,REG.
864

```

```

865 *****
866

```

```

procedure setreq(length: integer; value: real; index: integer);

```

```

var
val: real;
ind, k, j, l, upper, low: integer;

```

```

begin(*l*)
ind := index;
val := value;
k := 0;
upperlow(index, length, upper, low);
l := length;(*length of register*)
setindex(l, ind);(* if length = 0 then becomes = 1*)

```

```

903 k := low;
904
905 if length >= 2 then
906     ind := ind + length - 1; (*offset in long reg*)
907     for j := 1 downto 1 do begin(*2*) (*once if length = 0 or 1 *)
908         (*twice if the length = 2*)
909     repeat
910         r(ind)[k] := (val / 2 - trunc(val / 2)) * 2 = 1;
911         val := trunc(val / 2); (*transforms the decimal to binary*)
912         k := k + 1
913     until k > upper;
914     ind := ind - 1; (*in case of double register*)
915     k := 0
916 end; (*2*)
917 end; (*set reg 1*)

```



```

961
962
963
964
965 (*****
966
967 PROCEDURE retvalrea.
968
969 ----- :The procedure returns the decimal value of the register
970 Because the expected values of the register may exceeds the
971 the maximum permitted integer number of the UNIX system
972 the procedure uses real numbers.
973
974 PARAMETERS
975 ----- : by reference
976
977 - value is the returned value.The calling pro-
978 cedure is responsible to check if the number
979 is negative.Range is 0..2 to the power of 31
980 length is the length of the register.Range is
981 0..2.
982 - index is the number of the register.Range
983 is 0..23.
984
985 GLOBAL VARIABLES
986 ----- : -'r' represents the used register.
987 -R1f restricts the range of the procedure in the
988 four LSB of the register.Range is T or F.
989
990 LOCAL VARIABLES
991 -----
992 - The parameter value.
993 - ind is a copy of the parameter index.Range is 0..23.
994 - k is used as a bit index in the used register.
995 Range is 0..15.
996 - j is used as an index in the for loop.Range is 1..2.
997 - upper,low denotes the MSB and the LSB of the regi-
998 ster.Range of upper is 15 or 7, for low is 0 or 8.
999 - base denotes the base of the binary arithmetic
1000 system.
1001 The procedure uses copies of the parameters
1002

```

because the called procedures upperlow and set-index are called by reference and the procedure itself changes the value of the register's index.

The procedure calls : the procedures upperlow , setindex.  
The procedure is called by :DAORX,IMORIR,TRAP,LPDS,PUTATESHIFT,  
bLUCIRANSFER,outputreg,INPUNOUTPUT,BILMAN,pushstack,CALLRET,  
DIVMULT,ARITHMETIC,LOGICAL,compl,neg,MULTIOPER,EX,LOAD,STORE,  
REGISTER,REG.

\*\*\*\*\*)

procedure retvalreg(lenath, index: integer;  
var value : real );

var

base: real;

ind, k, j, l, upper, low: integer;

begin(\*1\*)

base := 1;(\*2 in the zero power \*)

ind := index;(\* index of the register\*)

value := 0;

upperlow(ind, lenath, upper, low);

l := length;(\*length of register in words\*)

setindex(l, ind);

if length >= 2 then (\*offset of register in words\*)

ind := ind + length - 1;

k := low;

if R11 then

upper := low + 3 + length;

for j := 1 downto 1 do begin(\*2\*)

repeat

if r(ind)k1 = true then

value := value + base;

base := base \* 2;(\*increment the power of 2 by one\*)

```

1081
1082
1083
1084
1085      k := k + 1
1086      until k > upper;      (*MSB of the register*)
1087      ind := ind - 1; (*for double words registers.*)
1088      k := 0
1089      end; (*2*)
1090      end; (*retval reg 1*)
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122

```



```

1201
1202
1203
1204
1205
1206  (*****
1207
1208
1209  PROCEDURE setmem.
1210
1211  FUNCTION
1212  ----- :Sets the memory to a decimal value transforming the
1213           value to hexadecimal one.Because the expected values
1214           may exceed the maximum integer of the UNIX system the
1215           procedure uses real numbers.The procedure checks if the
1216           address of the memory to be set is in the permitted
1217           range 0..maxmem.The Z-8000 does not check for this
1218           situation but it makes modulo operation and the user
1219           does not know if he has no correct address.
1220
1221  PARAMETERS
1222  ----- : by reference - None.
1223           : by value    - index denotes the address to be set.
1224           :               Range 0.. maxmem.
1225           :               - lbyte specifies if it is low or high
1226                           byte in the address.
1227                           Range is 0..1.
1228                           ex:
1229                           -----
1230                           address -! low byte! high byte!
1231                           -----
1232           :               - length denotes how many half bytes off-
1233                           set from the base address the procedure
1234                           will be set.Range is 0..8.
1235           :               - value is the decimal value.
1236                           Range is 0..2 to the power of 31.
1237
1238  GLOBAL VARIABLES
1239  -----:
1240           - memory is the array of the used real memory.
1241           - alldone denotes if the user's program.
1242             must suspend or not.Range is T or F.

```

1264  
1265  
1266  
1267  
1268  
1269  
1270  
1271  
1272  
1273  
1274  
1275  
1276  
1277  
1278  
1279  
1280  
1281  
1282  
1283  
1284  
1285  
1286  
1287  
1288  
1289  
1290  
1291  
1292  
1293  
1294  
1295  
1296  
1297  
1298  
1299  
1300  
1301  
1302  
1303  
1304  
1305  
1306  
1307

# LOCAL VARIABLES

```

-----: - val is a copy of the parameter value.
          Range is 0 .. 2 to the power of 31.
- ind is a copy of the parameter index
          Range is 0 .. maxmem.
- byt is a copy of the parameter lbyte.
          Range is 0..1.
- temp is a temporary variable used to
  transform the mod result to hex characters.
- j is used as index in the for loop.
          Range is 0..8.
  The procedure uses her mod function because
  it uses real numbers.The trasformation of
  mod result to hex characters is executed by
  the built in function chr.

```

The procedure calls: The procedure error.  
 The procedure is called by: BLOCKTRANSFER, TRANSFAFF, INPUTOUTPUT,  
 pushstack, ARITHMETIC, compl, neg, tset, MULTIOPER, LOAD STORE, EX, SIOKF.

\*\*\*\*\*

```

procedure setmem(index: real; lbyte, length: integer;
  value : real );

```

```

var
val: real;
ind, byt, temp, j: integer;

begin(*1*)
val := value; (*copy the value to the variable val*)
if length > 2 then(*more than one cell of memory must be set*)
  ind := trunc(index) + length div 2 - 1(*offset from *)
  (* base address*)

```



```

else
    ind := trunc(index);
if ind - (lenath div 2) < -1 then
    error(3); (*check if the address is less than the low bound*)
if ind > maxmem then
    error(5);(*check if the address is greater than the upper bound*)
byt := lbyte; (*copy the lbyte to the variable byt*)
if not alldone then (*if no error condition then*)
    for j := length downto 1 do begin(*2*)
        temp := trunc(val / hexv - trunc(val / hexv)) * hexv);
        (*finds the mod ex : temp= val mod 16*)
        if temp < 10 then (*digit 0..9*)
            temp := temp + 48
        else (*character A..F*)
            temp := temp + 55;
        memory[ind] [byt] := chr(temp);(*set the memory*)
        val := trunc(val / hexv);(*decreases the value*)
        byt := byt - 1;
        if byt < 0 then begin(*3*)
            byt := 1;
            ind := ind - 1 (*decrement offset from the base address*)
        end(*3*)
    end;(*2*)
end;(*1 setmem*)

```

```

1384 (*****
1385
1386
1387 PROCEDURE map.
1388
1389 FUNCTION
1390 ----- Returns the hexadecimal value of the memory transforming
1391 that in decimal one. The procedure uses real numbers because
1392 the expected values may exceed the maximum permitted in-
1393 teger from the UNIX system.
1394 Also it checks if it exists memory violation and then
1395 suspends the execution.
1396
1397 PARAMETERS
1398 ----- : by value - index is the base address to start the
1399 transformation. Range 0..maxmem.
1400 - lbyte denotes low or high byte in the
1401 address. Range is 0..1.
1402 - length specifies how many half bytes from
1403 the base address will be transform.
1404 Range is 0..8.
1405 by reference - res is the returned value. Range
1406 0 ..2 to the power of the 31.
1407
1408 GLOBAL VARIABLES
1409 -----
1410 - alldone denotes if the user's program
1411 must suspend or not.
1412 - memory is the array of the real used memory.
1413
1414 LOCAL VARIABLES
1415 -----
1416 - ind is a copy of the index .Range is 0..maxmem.
1417 - byte is a copy of the lbyte .Range is 0..1.
1418 - j is used as an index in the for loop. Range
1419 is 0..8.
1420 - mul is the base of hexadecimal numbers.
1421 The procedure uses copies for the parameters
1422 because changes the original values.
1423
1424
1425
1426
1427

```

```

1441
1442
1443
1444
1445 The procedure calls : The procedure error, valuechar
1446 the procedure is called by : DAORX,IMORIR,setsrdsst,LFDS,
1447 ROTAI,FSHIFT,compare,BLOCKTRANSFER,TRANSLATE, INPUTOUTPUT,BITMAN,
1448 CALLREI,DIVMULT,APITHMETIC,LOGICAL,compare,MULTIOPFR,LOAD,
1449 STORE.
1450
1451 *****
1452
1453 procedure map(index: real; lhbyte, length: integer;
1454           var res : real);
1455
1456   var
1457     j, k, ind, byt: integer;
1458     mul: real;
1459
1460   begin(*1*)
1461     mul := 1;(*base of hexadecimal to the zero power *)
1462     res := 0;(*initialize the result to zero*)
1463     if length > 2 then (*calculates the offset in the mem.*)
1464       ind := trunc(index) + length div 2 - 1
1465     else
1466       ind := trunc(index);
1467     if (ind - (length div 2) ) < -1 then
1468       error(3);(*checks if the address is lower than the low bound*)
1469     if ind > maxmem then
1470       error(5);(*checks if the address is over the upper bound*)
1471     byt := lhbyte;(*0 for low byte, 1 for high*)
1472     if not alldone then(*if no error condition has occurred then*)
1473       for j := length downto 1 do begin(*3*)
1474         valuechar(memory[ind],byt1,k);
1475         res := res + k * mul;
1476         mul := mul * 16; (*increases the power of 16 by one*)
1477         byt := byt - 1;
1478         if byt < 0 then begin(*4*)
1479           byt := 1;
1480           ind := ind - 1 (*decrement the offset*)
1481
1482

```

```
1504      end(*3*)
1505      end?(*4*)
1506      end:(*1 map *)
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
```

```

1561
1562
1563
1564
1565 (*****
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602

*****
PROCEDURE setflags.

FUNCTION
----- Sets the flags Z,S,PV of the RFC register after an ADD
or a logical operation.The calling procedure is responci-
ble to change the flags in the case of subtraction

PARAMETERS
----- by value - 'a' denotes if the result of the operation is
equal to zero.Range is T or F.
- ind denotes the register number.Range is 1..23.
- upper denotes the MSB of the register.Range
is 15 or 7.
- length denotes the length of the register.
Range is 0..4.
The procedure increases the index number because
the calling procedure pass the number of the
register increased by one.

GLOBAL VARIABLES
----- : - 'r' is the used register

LOCAL VARIABLES
----- : - None.

The procedure calls : None.
The procedure is called by : onetwocompl,ORANDXOR,ADD.
*****
procedure setflags(a: boolean; ind, upper, count,
length : integer );

begin(*1*)

```

1623  
1624  
1625  
1626  
1627  
1628  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649  
1650  
1651  
1652  
1653  
1654  
1655  
1656  
1657  
1658  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667

```
rffcwl[z] := not a;(*if a is false then zero flag*)
(*becomes true*)
rffcwl[s] := rind + 1[upper];
if length = 0 then
  rffcwl[pv] := not odd(count)(*checks for parity*)
end;(*1 setflag*)
```



```

1681
1682
1683
1684
1685 *****
1686 *****
1687 *****
1688 *****
1689 *****
1690 *****
1691 *****
1692 *****
1693 *****
1694 *****
1695 *****
1696 *****
1697 *****
1698 *****
1699 *****
1700 *****
1701 *****
1702 *****
1703 *****
1704 *****
1705 *****
1706 *****
1707 *****
1708 *****
1709 *****
1710 *****
1711 *****
1712 *****
1713 *****
1714 *****
1715 *****
1716 *****
1717 *****
1718 *****
1719 *****
1720 *****
1721 *****
1722 *****

PROCEDURE onetwocompl.

----- the procedure makes three functions depending on the values
of the parameters
- one complement of a specified register changing also
  the flags.
- two complement of a specified register changing also
  the flags.
- one or two complement of a specified register without
  to change the flags.

PARAMETERS
-----:by value - index is the number of the register.Range
is 0..23.
- length is the length of the register.Range
  0 is 0..4.
- oper denotes the type of the operation
  1 denotes one complement.
  2 denotes two complement.
- change denotes if the flags will be changed
  after the operation.Range is 1 or F.

GLOBAL VARIABLES
-----: - 'a' is a temporary boolean variable used to check
if the result is zero.Range is T or F.
- 'r' is the used register.

LOCAL VARIABLES.
-----: - first is a boolean variable that is set to true
if the operation is two complement and it is
changed to false if the first one hit in the
source register is encountered and the operation
changes to one complement.

```

1743  
1744  
1745  
1746  
1747  
1748  
1749  
1750  
1751  
1752  
1753  
1754  
1755  
1756  
1757  
1758  
1759  
1760  
1761  
1762  
1763  
1764  
1765  
1766  
1767  
1768  
1769  
1770  
1771  
1772  
1773  
1774  
1775  
1776  
1777  
1778  
1779  
1780  
1781  
1782  
1783  
1784  
1785  
1786  
1787

```

ex 00011000 in two complement
the last three bits will be remain unchanged
and then the operation will be changed
to one complement.
- count checks for even parity in the case that the
operation is executed in a byte register.
- j is used as index in the for loop.Range is 0..4.
- l is a copy of the length parameter .Range is 0..4.
- k is set to the LSB at each iteration of the for
loop.Range is 0 or 8.
- ind is a copy of the original value of the index
and it is changed during the operation depending
on the length of the register.Range is 0..23.
ex if the passed value for the register is 0 and
the length is 4 the the operation will start
from the lowest register of the R00 which is
the R3 and the will continue to the registers
R2 , R1 , R0 .
- upper,low denotes the MSB and the LSB of the
register.
The LSB of the register it is changed during
the operation and the variable k is used as a copy
of the original value.

```

```

the procedure calls : the procedures upperlow,setindex,setflags.
the procedure is called by : compare,positive,DIVMUL1,ARITHMETIC,
compare,neg,MULTIOPFR,compl.

```

```

*****

```

```

procedure onetwocompl(index, length, oper: integer;
change : boolean );

```

```

var
first: boolean;

```

```

1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842

count, j, k, ind, l, upper, low: integer;

begin(*1*)
  first := oper = 2;(*check if operation is two complement*)
  a := false;(*check for zero*)
  count := 0;(*check for even parity*)
  ind := index;
  upperlow(ind, length, upper, low);
  l := length;
  setindex(l, ind);
  k := low;
  if length >= 2 then
    ind := ind + length - 1;(*calculates the offset of the *)
    (* register for the case of long registers*)
    for j := 1 downto l do begin(*2*)
      repeat
        if first then
          first := r[ind][k] = false(*search to find the first one *)
          (*bit in the case of two complement*)
        else
          r[ind][k] := not r[ind][k]; (*one complement or two after*)
          (*the first one has been encountered*)
          a := a or r[ind][k];(*check if all bits are zero*)
          if r[ind][k] then
            count := count + 1;(*then check for parity*)
            k := k + 1;
            until k > upper;
            ind := ind - 1;
            k := 0
          end;(*2*)
          if change then (*if the change is true*)
            setflags(a, ind, upper, count, length)
          end;(* 1 onetwocompl*)

```

1864  
1865  
1866  
1867  
1868  
1869  
1870  
1871  
1872  
1873  
1874  
1875  
1876  
1877  
1878  
1879  
1880  
1881  
1882  
1883  
1884  
1885  
1886  
1887  
1888  
1889  
1890  
1891  
1892  
1893  
1894  
1895  
1896  
1897  
1898  
1899  
1900  
1901  
1902  
1903  
1904  
1905  
1906  
1907

PROCEDURE testcc.

FUNCTION  
-----

The procedure checks for required conditions to the flags of the RFC register and returns an answer true or false. The required conditions are passed by parameter cccode and the answer is returned by the parameter cc.

The check conditions are the following:

CODE	MEANING	FLAG SETTINGS	BINARY
---	----	-----	-----
	Always false	--	0000
	Always true	--	1000
Z	Zero	Z=1	0110
NZ	Not zero	Z=0	1110
C	Carry	C=1	0111
NC	No carry	C=0	1111
PL	Plus	S=0	1101
MI	Minus	S=1	0101
NE	Not equal	Z=0	1110
EQ	Equal	Z=1	0110
OV	Overflow	V=1	0100
NOV	No overflow	V=0	1100
GE	Greater than or equal	(S XOR V) = 0	1001
LT	Less than	(S XOR V) = 1	0001
GT	Greater than	(Z OR (S XOR V)) = 0	1010
LE	Less than or equal	(Z OR (S XOR V)) = 1	0010
UGL	Unsigned greater than	C = 0	1111

```

1921
1922
1923
1924
1925
1926      or equal
1927      unsigned
1928      less than
1929      unsigned
1930      greater than
1931      unsigned less
1932      than or equal
1933
1934      C = 1
1935      0111
1936
1937      ((C=0) AND (Z=0)) = 1
1938      1011
1939
1940      (C OR Z) = 1
1941      0011
1942
1943      PARAMETERS
1944      -----
1945      By value - cccode is the condition to be checked.
1946      Range is 0..15
1947
1948      By reference - cc is the answer . Range is T or F.
1949
1950      GLOBAL VARIABLES
1951      -----
1952      - The parameter cc.
1953      - code is a copy of the cccode in integer, because
1954      the cccode may be passed as a real value.
1955      Range is 0..15
1956
1957      The procedure calls : None
1958
1959      The procedure is called by: compare, CALLRET, ARITHMETIC
1960
1961      *****
1962      procedure testcc(var cc: boolean; cccode: real);
1963      var
1964      code: integer; (x1x)
1965      begin
1966      code := trunc(cccode);
1967      case code of
1968      0:
1969
1970
1971

```

```

cc := false; (*always false*)
1:
cc := r[fcw][s] and not r[fcw][pv]
    or r[fcw][pv] and not r[fcw][s]; (*LI *)
(* (S XOR V) = 1 *)
2:
cc := r[fcw][z] or (r[fcw][s] and not r[fcw][pv]
    or r[fcw][pv] and not r[fcw][s]); (*LE *)
(* Z or (S XOR V) = 1 *)
3:
cc := r[fcw][c] or r[fcw][z] = true; (*ULE*)
(* (C OR Z) = 1 *)
4:
cc := r[fcw][pv] = true; (*UV*)
(* PV = 1 *)
5:
cc := r[fcw][s] = true; (*MI*)
(* S = 1 *)
6:
cc := r[fcw][z] = true; (*EQ or zero*)
(* Z = 1 *)
7:
cc := r[fcw][c] = true; (*UI or carry*)
(* C = 1 *)
8:
cc := true; (*always true*)
9:
cc := r[fcw][s] and not r[fcw][pv]
    or not r[fcw][s] and r[fcw][pv] = false; (*GF *)
(* (S XOR V) = 0 *)
10:
cc := r[fcw][z] or (r[fcw][s] and not r[fcw][pv]
    or r[fcw][pv] and not r[fcw][s]) = false; (*GF*)
(* Z OR (S XOR V) = 0 *)
11:

```



```

2041 cc := (r[fcw][c] = false) and (r[fcw][z] = false) = true; (*UGT*)
2042 (* (( C = 0 ) AND ( Z = 0 ) ) = 1 *)
2043 12:
2044 cc := r[fcw][pv] = false; (*NUV*)
2045 (* PV = 0 *)
2046 13:
2047 cc := r[fcw][s] = false; (*plus*)
2048 (* S = 0 *)
2049 14:
2050 cc := r[fcw][z] = false; (*INE or no zero*)
2051 (* 7 = 0 *)
2052 15:
2053 cc := r[fcw][c] = false
2054 (* C = 0 *)
2055 end (*case*)
2056 end; (* testcc *)

```



```

2161         upper2 are 7 or 15.
2162     - 11,12 are copy of the parameter length.Range
2163       is 0..4.
2164     - ind1,ind2 are copy of the parameters dst and
2165       src.Range is 0..23.
2166     - count is used to check for even parity in the
2167       case of byte register.
2168
2169 the procedure calls : the procedures upperlow,setindex,setflags.
2170 the procedure is called by : LOGICAL ,MULTIOPK.
2171
2172 *****
2173 *****
2174 *****
2175 *****
2176 *****
2177 *****
2178 *****
2179 *****
2180 *****
2181 *****
2182 *****
2183 *****
2184 *****
2185 *****
2186 *****
2187 *****
2188 *****
2189 *****
2190 *****
2191 *****
2192 *****
2193 *****
2194 *****
2195 *****
2196 *****
2197 *****
2198 *****
2199 *****
2200 *****
2201 *****
2202 *****

    procedure GRANDXUR(dst, src, length, value: integer;
        change : boolean );

var
j, low1, low2, upper1, upper2, l1, l2, ind1, ind2,
k1, k2 , count : integer;

begin(*1*)
a := false;(*check for zero*)
count := 0;(*check for even parity*)
ind1 := dst;(*copies the number of dst register*)
l1 := length;(*copies the length of the dst register*)
upperlow(ind1, length, upper1, low1);
setindex(l1, ind1);
if value <> 0 then begin(* two registers for comparison*)(*2*)
    ind? := src;(*copies the number of the second register*)
    l2 := length;(*copies the length of the second register*)
    upperlow(ind2, length, upper2, low2);

```

```

2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267

    setindex(l2, ind2)
end?(*2*)
if l1 >= 2 then begin(*3*)
    ind1 := ind1 + length - 1;(*offset in case of long *)
    (* registers *)
    ind2 := ind2 + length - 1
end?(*3*)
k1 := low1;
k2 := low2;
for j := 11 downto 1 do begin(*4*)
    repeat
        case value of
            0:
                rfind1[k1] := rfind1[k1] or false;(*Rdst OR 0*)
            1:
                rfind1[k1] := rfind1[k1] or rfind2[k2];(*Rdst←Rdst *)
                (* OR Rsrc *)
            2:
                rfind1[k1] := rfind1[k1] and rfind2[k2];(*Rdst←Rdst *)
                (* AND Rsrc *)
            3:
                (*Rdst ← Rdst XOR Rsrc*)
                rfind1[k1] := rfind1[k1] and not rfind2[k2]
                    or rfind2[k2] and not rfind1[k1]
        end? (*acase*)
        a := a or rfind1[k1];(*check if all the bits are zero*)
        if rfind1[k1] then
            count := count + 1;(*check for even parity*)
            k1 := k1 + 1;
            k2 := k2 + 1
        until k1 > upper1;
        k1 := 0;
        k2 := 0;
        ind1 := ind1 - 1;
        ind2 := ind2 - 1
    end?
end?

```

```
2281
2282
2283
2284
2285      end;(*4*)
2286      if change then
2287          setflags(a, ind1, upper1, count, length)
2288      end;(*1 0RANDOMXOR*)
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
```

```

2343 (*****
2344
2345 *****
2346 *****
2347 *****
2348 *****
2349 *****
2350 *****
2351 *****
2352 *****
2353 *****
2354 *****
2355 *****
2356 *****
2357 *****
2358 *****
2359 *****
2360 *****
2361 *****
2362 *****
2363 *****
2364 *****
2365 *****
2366 *****
2367 *****
2368 *****
2369 *****
2370 *****
2371 *****
2372 *****
2373 *****
2374 *****
2375 *****
2376 *****
2377 *****
2378 *****
2379 *****
2380 *****
2381 *****
2382 *****
2383 *****
2384 *****
2385 *****
2386 *****
2387 *****

```

PROCEDURE ADD

FUNCTION

-----

The procedure performs binary addition in two registers of any length using the following formula

SUM = A XOR B XOR C  
CARRY = AR + AC + BC

Also performs the following functions :

- Sets the half carry if it exists in the case of ADDB or ADDBC operation
- Checks for zero result.
- Checks for overflow in the case that the destination and source register have the same sign.
- Sets the flags depending on the result, type of the register and if the parameter change is 1 or F.

PARAMETERS

-----

By value      - dst,src are the destination and source registers  
                     Range 0..23

                    - length is the length of the register.  
                     Range is 0..4.

                    - change denotes if the flags will be changed.  
                     Range is 1 or F.

                    - car denotes if the addition will be executed with carry or not. It is true in the case of the opcode ADC or SUBC.

LOCAL VARIABLES



```

2401
2402
2403
2404
2405 -----
2406 - low1, low2, upper1,upper2 denote the MSB and
2407   LSB of the destination and source registers.
2408   Range for low1 is 0 or 7 , range for upper1 is 8
2409   or 15.
2410 - 11,12 are copy of the parameter length. Range is
2411   0..4.
2412 - k1,k2 are used as index of the bit of the registers.
2413   Range is 0..15.
2414 - ind1, ind2 are copy of the parameters dst, src.
2415   Range is 0..23
2416 - j is used as index in the for loop. Range is 1..4
2417 - check if true denotes that the destination and
2418   source register before the operation have the same
2419   sign
2420 - carry is the produced carry or copy of the existed
2421   carry in the case of ADDC or SURC operation.
2422 - precond if the sign of the two registers is the same
2423   then it saves the sign of the destination register.
2424   Range is T or F.
2425 - half is the half carry .Is used only in operation
2426   with byte registers.Range is T or F.
2427 - zero denotes if the result is zero.Range is T or F.
2428

```

The procedure calls :The procedures upperlow,setindex,setflags.

The procedure is called by : compare,DAB,ARITHMETIC,compare,LOAD.

\*\*\*\*\*)

```

procedure ADD(dst, src, length: integer; change,
              car : boolean );

```

```

2462 var
2463 l1, l2, low1, low2, upper1, upper2, k1, k2, ind1, ind2, j: integer;
2464 check, carry, precond, half, zero: boolean;
2465
2466 begin(*1*)
2467 zero := false;
2468 if car then (*ADD with carry*)
2469     carry := r[fcw]fcl(*if carry exists*)
2470 else
2471     carry := false;
2472 ind1 := dst;(*copies the numbers of the dst and src registers*)
2473 ind2 := src;
2474 l1 := length;(*copies the length of the dst and src registers*)
2475 l2 := length;
2476 upperlow(dst, l1, upper1, low1);
2477 upperlow(src, l2, upper2, low2);
2478 setindex(l1, ind1);
2479 setindex(l2, ind2);
2480 k1 := low1;
2481 k2 := low2;
2482 if r[ind1][upper1] = r[ind2][upper2] then begin(*2*)
2483     check := true;(*in the case when the dst and src*)
2484     (*have the same sign checks for overflow*)
2485     precond := r[ind1][upper1](*save the sign*)
2486 end;(*2*)
2487 if l1 >= 2 then begin(*3*)
2488     ind1 := ind1 + length - 1;
2489     ind2 := ind2 + length - 1
2490 end;(*3*)
2491 for j := l1 downto 1 do begin(*4*)
2492     repeat
2493         a := r[ind1][k1];
2494         b := r[ind2][k2];
2495         r[ind1][k1] := not a and b or a and not b;

```

```

2521      j := r[indl][k1];(*the addition is executed in two steps*)
2522      r[indl][k1] := not d and carry or d and not carry;
2523      carry := a and b or a and carry or b and carry;
2524      if (length = 0) and (k1 = 3) then
2525          half := carry;(*half carry for the case of ADDB or ADDbC*)
2526      zero := zero or r[indl][k1];(*if all the bits are zero*)
2527      k1 := k1 + 1;
2528      k2 := k2 + 1
2529      until k1 > upper1;
2530      k1 := 0;
2531      k2 := 0;
2532      ind1 := ind1 - 1;
2533      ind2 := ind2 - 1
2534      end;(*4*)
2535      if change then
2536          setflags(zero, ind1, upper1, 0, length);
2537          r[fcw][lcl] := carry;
2538      if check then (* in the case of the same sign *)
2539          (* checks for the sign in the result *)
2540          r[fcw][lv] := precond <> r[indl + 1][upper1](*overflow*)
2541      else
2542          r[fcw][lv] := false;
2543      if length = 0 then begin(*5*)
2544          r[fcw][lda] := false; (* for decimal adjust *)
2545          r[fcw][h] := half (* for half carry *)
2546      end(*5*)
2547      end;(*1 ADD*)

```

(\*\*\*\*\*  
2584  
2585  
2586  
2587  
2588  
2589  
2590  
2591  
2592  
2593  
2594  
2595  
2596  
2597  
2598  
2599  
2600  
2601  
2602  
2603  
2604  
2605  
2606  
2607  
2608  
2609  
2610  
2611  
2612  
2613  
2614  
2615  
2616  
2617  
2618  
2619  
2620  
2621  
2622  
2623  
2624  
2625  
2626  
2627

PROCEDURE DAORX

FUNCTION

-----  
Returns the address of the opcode either in the case of  
direct address ofr int the case of indexed address  
The existed two conditions are the following

Direct address

dst = 0

-----  
! opcode ! src or dst ! dst or src !  
-----

address

-----  
Index address

dst <> 0

-----  
! opcode ! src or dst ! dst or src !  
-----

direct part

-----  
final address : = contents Rx + direct part  
Also it increments the counter.

PARAMETERS

-----  
by value: - dst is the value of the destination part of the  
of the opcode . Range is 0..15

```

2641      by reference: - val is the computed address. The pro-
2642                    cedure does not check if the address
2643                    is correct.
2644
2645      GLOBAL VARIABLES
2646      -----
2647      - ic is the decimal counter that has a copy of the
2648        value of the RFC .Range is 0 ..maxmem.
2649
2650      LOCAL VARIABLES
2651      -----
2652      - value is a temporary variable used in the case
2653        of indexed address. The procedure does not check
2654        if the contents of the register are correct.
2655      - The parameter val.
2656
2657      The procedure calls : The procedures map,retvalreg.
2658
2659      The procedure is called by: ROTATESHIFT,BITMAN,CALLRET,DIVMULT,
2660      ARITHMETIC,LOGICAL,EX,LOAD,STORE.
2661
2662      *****
2663      procedure DAorX(dst: integer; var val: real);
2664
2665      var
2666      value: real;
2667
2668      begin(*1*)
2669      value := 0;
2670      map(ic + 1, 1, 4, val); (* direct part of the address *)
2671      if dst < 0 then (*no direct address*)
2672          retvalreg(1, dst, value);(*value ← [Rx]*)
2673      val := val + value;(* address plus the contents of the register*)
2674
2675
2676
2677
2678
2679
2680
2681
2682

```

```
ic := ic + 3  
end:(*1 UaorX*)
```

2704  
2705  
2706  
2707  
2708  
2709  
2710  
2711  
2712  
2713  
2714  
2715  
2716  
2717  
2718  
2719  
2720  
2721  
2722  
2723  
2724  
2725  
2726  
2727  
2728  
2729  
2730  
2731  
2732  
2733  
2734  
2735  
2736  
2737  
2738  
2739  
2740  
2741  
2742  
2743  
2744  
2745  
2746  
2747



```

2761
2762
2763
2764
2765 (*****
2766
2767
2768
2769
2770 FUNCTION
2771 -----
2772
2773 The procedure returns the value of the data in either case
2774 of immediate ones or indirect ones.
2775 The data are IM or IR depending on the value of source
2776 register of the opcode.If the value of src = 0 then the data
2777 are IM.
2778 So the procedure in the case of IM data transforms the
2779 contents of the memory that follows the opcode in decimal
2780 value. Else it finds the contents of the specified register
2781 and using them as address finds the data.
2782
2783 Also it increments the counter depending on the type of data
2784 and the type of opcode if it is byte or word or double word.
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802

```

PROCEDURE IMorIP

by value - src is the value of the source register.  
 Range is 0..15.If = 0 then IM data.

- 1 specifies the number of half bytes to be trans-  
 formed from the memory to decimal value.Range  
 is 2 or 4 or 8 depending on the type of the op-  
 code.

- length is the length of the register  
 Range is 0..4.

by reference - val is the returned result

GLOBAL VARIABLES

-----

-ic is the decimal counter

```

2824 LOCAL VARIABLES
2825 -----
2826
2827 -The parameter val .Range is 0..2 to the power of 31
2828 - value is used to save the contents of the register
2829 that is used in the IR mode .No check in the range.
2830 is done.
2831
2832 The procedure calls: The procedures map, retvalreq,
2833
2834 The procedure is called by: DIVMULT,ARITMETIC,LOAD,STORE.
2835
2836 *****
2837
2838
2839
2840
2841 procedure IMorIR(src, l, length: integer; var val: real);
2842
2843 var
2844 value: real;
2845
2846 begin(*l*)
2847 (* -----
2848 ; opcode ; src ; dst ;
2849 -----
2850 ; IM value ;
2851 -----
2852
2853 ; IM value ;
2854 -----
2855
2856 (* if src <> 0 then
2857 -----
2858 ; opcode ; src ; dst ;
2859 -----
2860
2861
2862
2863
2864
2865
2866
2867

```

if src = 0 then the half upper  
part if is byte opcode or the  
whole part if one word opcode.  
in case of opcode for double  
register

```

2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921

-----
*)

if src = 0 then (*1M value*)
    map(ic + 1, 1, val)(*val ← 1M*)
else begin(*2*)(*then indirect value *)
    retvalreg(1, src, value);(*value ← [Rsrc]*)
    map(value, 1, 1, val)
end;(*val ← mem[value])(*2*)
if src = 0 then begin(*3*)
    if length = 2 then
        ic := ic + 5(*double register*)
    else
        ic := ic + 3(*single word register*)
    end else(*3*)
        ic := ic + 1(*only IR address *)
end; (* 1 [MorIR*)

```

```

2944  (*****
2945  *****
2946  *****
2947  *****
2948  *****
2949  *****
2950  *****
2951  *****
2952  *****
2953  *****
2954  *****
2955  *****
2956  *****
2957  *****
2958  *****
2959  *****
2960  *****
2961  *****
2962  *****
2963  *****
2964  *****
2965  *****
2966  *****
2967  *****
2968  *****
2969  *****
2970  *****
2971  *****
2972  *****
2973  *****
2974  *****
2975  *****
2976  *****
2977  *****
2978  *****
2979  *****
2980  *****
2981  *****
2982  *****
2983  *****
2984  *****
2985  *****
2986  *****
2987  *****

PROCEDURE setlength)

FUNCTION
----- The procedure specifies the length of the used registers
          during the execution of any opcode and this is depended
          from the type of the opcode if it is byte or one
          word or double word register opcode.
          Also it specifies how many half bytes will be needed by the
          operation in the case when it makes reference in the memory.

PARAMETERS
-----
          by reference - length is the returned value of the
                        length of the registers to the calling
                        procedure.Range is 0..2.
          - 1 is the number of the half bytes in the
            case that the procedure makes reference
            to the memory.Range is 2 or 4 or 8.

GLOBAL VARIABLES
-----: - opcode is the current opcode under execution.
          Range is 0 ..192,240,224,208.

LOCAL VARIABLES.
-----: - None.

The procedure calls : None.
The procedure is called by:ROTATESHIFT,BLOCKTRANSFER,INPUTOUTPUT,
BITMAN,DIVMULT,ARITHMETIC,LOGICAL,MULTIOPER,EX,LOAD,STORE.
*****

```

```

3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041

procedure setlength1(var length, l: integer);
begin(*1*)
case opcode of
0, 2, 4, 6, 8, 10, 12,
32, 34, 36, 38, 40, 42, 44,
46, 48, 50, 58, 60, 62, 64,
66, 68, 70, 72, 74, 76, 96,
98, 100, 102, 104, 106, 108, 110,
112, 114, 128, 130, 132, 134, 136,
138, 140, 160, 162, 164, 166, 168,
170, 172, 174, 177, 178, 180, 182,
186, 188, 190, 192:
begin(*2*)
length := 0;(*byte registers*)
l := 2 (* corresponds to two characters in memory *)
end;(*2*)
1, 3, 5, 7, 9, 11, 13,
19, 23, 25, 27, 33, 35, 37,
39, 41, 43, 45, 47, 49, 51,
52, 59, 61, 63, 65, 67, 69,
71, 73, 75, 77, 83, 87, 89,
91, 97, 99, 101, 103, 105, 107,
109, 111, 113, 115, 116, 118, 125,
129, 131, 133, 135, 137, 139, 141,
147, 151, 153, 155, 161, 163, 165,
167, 169, 171, 173, 175, 179, 181,
183, 187, 189:
begin(*3*)
length := 1;(*one word registers*)
l := 4 (* corresponds to four characters in memory *)
end;(*3*)
16, 17, 18, 20, 21, 22, 24,
26, 28, 29, 53, 55, 80, 81,

```

```

3064      82, 84, 85, 86, 88, 90, 92,
3065      93, 117, 119, 144, 146, 145, 148,
3066      149, 150, 152, 154, 156:
3067      begin(*4*)
3068          length := 2: (*double registers*)
3069          l := 8 (* corresponds to eight characters in memory *)
3070          end (*4*)
3071      end (*case*)
3072      end; (*1 setlength1*)
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3100
3101
3102
3103
3104
3105
3106
3107

```



```

3121
3122
3123
3124
3125
3126  (*****
3127
3128
3129      PROCEDURE setsrsrcdst
3130
3131  -----The procedure calculates the numbers of the source and
3132          destination register in the opcode.
3133
3134  PARAMFIFRS
3135  ----- by reference : -src is the source register.Range is 0..15.
3136          - dst is the destination register.Range
3137          is 0..15.
3138
3139  GLOBAL VARIABLES
3140  ----- : - ic is the current value of the counter.Range
3141          is 0 ..maxmem.
3142
3143  LOCAL VARIABLES
3144  ----- : - x is used as temporary variable to assign
3145          the values of the registers to the parameters
3146          dst and src.
3147
3148  The procedure calls : The procedure map
3149  The procedure is called by : ROTATESHIFT,BLOCKTRANSFER,TRANSLATE,LDCIL,
3150  BITMAN,MULTIUPER,CALLRET,STORF,EXFCUIF.
3151
3152  *****
3153
3154      procedure setsrsrcdst(var src, dst: integer);
3155      var
3156      x: real;(*1*)
3157      begin
3158      map(ic, 1, 1, x);
3159      dst := trunc(x);
3160
3161
3162

```

```
3184 map(ic, 0, 1, x);  
3185 src := trunc(x)  
3186 end;(*1 setsrcdst*)
```

```
3187  
3188  
3189  
3190  
3191  
3192  
3193  
3194  
3195  
3196  
3197  
3198  
3199  
3200  
3201  
3202  
3203  
3204  
3205  
3206  
3207  
3208
```

1  
2  
3  
4  
5  
6  
7 (\*\*\*\*\*  
8  
9

PROCEDURE DAB.

10 FUNCTION

11 ----- :The destination byte is adjusted to form two 4 bit BCD  
12 digits following an addition or subtraction operation  
13 For addition (ADDB ,ADCB) or subtraction (SUBB,SBCB),  
14 the following table indicates the operation performed:  
15

instruction	carry	bits 4..7	H flag	bits 0..3	number	carry
	before	value	before	value	added	after
	DAB	(HEX)	DAB	(HEX)	to byte	DAB
20 ADCB	F	0..9	F	0..9	00	F
21 ADCB	F	0..A	F	A..F	06	F
22	F	0..9	T	0..3	06	F
23	F	A..F	F	0..9	60	T
24	F	9..F	F	A..F	66	T
25	F	A..F	T	0..3	66	T
26	T	0..2	F	0..3	60	T
27	T	0..2	0	A..F	66	T

28						
29 SUBB	F	0..9	F	0..9	00	F
30 SBCB	F	0..A	T	6..F	FA	F
31	T	7..F	F	0..9	A0	T
32	T	6..F	T	6..F	9A	T

33 -----  
34 The operation is undefined if the destination byte was not the  
35 result of a valid addition or subtraction of BCD digits.

36 PARAMETERS

37 ----- : none  
38  
39  
40  
41

# GLOBAL VARIABLES

```

-----: -ADDB denotes that ADDB OR ADCB operation has
65      been executed .Range I or F.
66      - SUBB denotes that SUBB OR SBCB operation has
67      been executed .Range I or F.
68      ( The procedure ARITHMETIC sets the above flags)
69      - ic is the decimal counter .Range 0 maxmem.
70      -src is the dst operand .It is decoded by the
71      procedure EXECUTE .Range 0..15.
72
73
74

```

# LOCAL VARIABLES

```

-----: -number is the number that will be added to adjust
75      the result.Range 0,6,60,66,FA,A0,9A.
76      - val is the contents of the dst register.Range
77      0..255.
78      - low is the value of the half low byte of the dst
79      register(bits 0..3).Range 0..15.
80      - high is the value of the high half byte of the
81      destination register(bits 4..7).Range 0..15.
82      - value is a copy in integer of the variable val
83      and is used in the mod operation.Range 0..255.
84      - carry , hflag are copies of the flags C and H.
85      Range I of F.
86
87

```

```

88 The procedure checks first if ADDB has been executed else it jumps
89 in the SUBB case.
90 If the user has executed ADDB and SUBB then the system executes the
91 the last one operation because the procedure ARITHMETIC in each
92 execution resets the two flags to false.
93 The flags C,7,S are set according to the result of the operation and
94 the above table.
95 To make the adjustment the procedure uses the temporary register
96 R17 which is set to the value that will be added to the destination
97 register and then adds the R17 to the destination register without
98 to affect the flags.
99
100
101
102
103
104
105
106
107

```

```

121
122
123
124
125 The procedure calls : the procedures retvalreg, setreg.
126 The procedure is called by : EXECUTE.
127
128
129
130
131 *****
132 *****
133 *****
134 *****
135 *****
136 *****
137 *****
138 *****
139 *****
140 *****
141 *****
142 *****
143 *****
144 *****
145 *****
146 *****
147 *****
148 *****
149 *****
150 *****
151 *****
152 *****
153 *****
154 *****
155 *****
156 *****
157 *****
158 *****
159 *****
160 *****
161 *****

procedure DAB;

var
val: real;
number, low, high, value: integer;
carry, hflag: boolean;

begin (*1*)
number := 0; (*if it is not needed the operation DAB then *)
(* no number is added to the result *)
ic := ic + 1;
retvalreg(0, src, val);(*value of the register over which the *)
(* DAB operation will be executed*)
value := trunc(val);
low := value mod 16;(*value of bits 0..3*)
high := value div 16; (*value of bits 4..7*)
carry := rffcwlcl;(*value of carry before the operation*)
hflag := rffcwl[h]; (*value of the H flag before the operation*)
case ADDR of
false:
null; (*no operation*)
true:
begin (*2*)(*ADDR is true*)

```

```

184 case carry of
185 true:
186     begin (*3*)
187     case hflag of
188     true:
189         null; (*no operation*)
190     false:
191         begin (*4*)
192             if high <= 2 then begin (*5*)
193                 if low <= 9 then
194                     number := 60 (*the number that will be added *)
195                 (* to adjust the result in decimal *)
196             else
197                 number := 60
198             end(*5*)
199         end(*4*)
200     end (*case hflag*)
201 end; (*3 carry = true*)
202 false:
203     begin (*6*)
204     case hflag of
205     true:
206         begin (*7*)
207             if (high <= 9) and (low <= 3) then
208                 number := 6
209             end; (*7*)
210         false:
211             begin (*8*)
212                 (*if high <= 9) and (low <= 9) then null;*)
213                 if (high <= 8) and (low >= 10) then
214                     number := 6;
215                 if (high >= 10) and (low <= 9) then
216                     number := 60;
217                 if (high >= 9) and (low >= 10) then
218                     number := 60
219                 end
220             end
221         end
222     end
223 end
224
225
226
227

```



```

241         end (*8*)
242         end (*case hflag of false carry*)
243         end(*6*)
244     end ; (*case of carry*)
245     if (number = 66) or (number = 60) then
246         rlfcwjcl := true
247     end(*2*)
248     end ; (*case ADDB*)
249     case SUBB of
250     false:
251         null; (*no operation*)
252     true:
253         begin (*20*)
254             case carry of
255             true:
256                 begin (*21*)
257                     if hflag and (high >= 6) and (low >= 6) then
258                         number := 154;
259                     if not hflag and (high >= 7) and (low <= 9) then
260                         number := 160;
261                     rlfcwjcl := true
262                     end; (*21*)
263                 false:
264                     begin (*22*)
265                         if hflag and (high <= 8) and (low >= 6) then
266                             number := 250
267                         end(*22*)
268                     end(*case of carry*)
269                     (*if not hflag and high <= 9 and low <= 9 then null;*)
270                 end (*20*)
271             end; (*case SUBB*)
272             if number <> 0 then begin (*23*)
273                 setreg(0, number, temporary);(*K17 ← number to be added*)
274                 ADD(src, temporary, 0, false, false)
275                 end (*Rsrc ← K17 + Rsrc*); (*23*)
276             end
277         end
278     end
279
280
281

```

```
303  
304     AODB := false;  
305     SUBB := false  
306     end; (* 1 DAR *)  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347
```

361

362

363

364

365

366

367

368

# FUNCTION

-----

The procedure performs all the functions rotate and shift depending on the number of the opcode.

As nested procedures it uses the procedures :

- valuebytes that is used to calculate the half byte values of the used register in the case of execution of the instructions RPDR and RLDB.
- shifttrintleft that is used as the tool to make the shifts.

376

377

378

379

380

381

382

383

384

385

386

387

388

389

390

391

392

393

394

395

396

397

398

399

400

401

## PROCEDURE ROTATESHIFT

### GLOBAL VARIABLES

-----

- 'r' is the used register.
- opcode is the current opcode under execution.

### LOCAL VARIABLES

-----

- val,value, temp are used as temporary variables to save and send values to other procedures as parameters.Range 0..maxinteg.
- dst is the destination register .Range 0..15.
- l denotes the number of half bytes to be trasformed from the memory to decimal value.Range 2,4,or8.
- length denotes the length of the register.Range 0..2.
- upper,low are the MSB and LSB of the used register. Range upper 8 or 15 and low range 0 or 7.
- mshdst is the value of dst register for the bits 4..7. Range 0..15.
- lshdst is the value of the dst register for the bits 0..3.Range 0..15.
- mshsrc is the value of src register for the hits 4..7.

- Range 0..15.  
lshsrc is the value of src register for the bits 0..3.  
Range 0..15.  
(The above 4 variables are used only in the case of execution of the instructions RLDR or RRDR.)
- times denotes how many times the opcode will be executed for the cases when the opcode is repeat type. Range is +- 8 for byte registers, +- 16 for one word registers and +- 32 for double word registers.
- select denotes the type of the opcode because all the RL,RPC opcodes have the same value in the first byte of the opcode but they differ only in the last four bits of the word of the opcode.
- right denotes that is right or left rotation. Range is L or F.
- negative denotes that either the contents of the register that has the value of the times that opcode will be executed is negative or the IM data are negative and so the rotation is right. Range is l or F.
- correct denotes if the specified number of rotations are correct or the number of the register is in the permitted range 0..15. If not correct then the opcode is not executed as it is specified in the Z-8000 manual. Range is l or F.

The procedure calls : The procedures setsrcdst ,setlenathl,retvalreg.  
valuebytes,setren,shiftrightleft.

The procedure is called by : LEXECUTE.

\*\*\*\*\*)



\*\*\*\*\*

```
procedure valuebytes(value, val: real);
```

begin (★1★)

```

msbdst := trunc(value) div 16;(*msb of dst register*)
lsbdst := trunc(value) mod 16;(*lsb of dst register*)
msbsrc := trunc(val) div 16;(*msb of src register*)
lsbsrc := trunc(val) mod 16;(*lsb of src register*)

end; (*1 valuebytes*)

```

```
end; (*1 valuebytes*)
```



```

601
602
603
604
605  (*****
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641

```

PROCEDURE shiftrightleft

FUNCTION

Is a multipurpose procedure depending on the values of the parameters n and right as follows :

n	right	operation
1	T	SDA (B,L) RIGHT
1	F	SDA (B,L) LEFT
2	F	SDL (B,L) LEFT
2	T	SDL (B,L) RIGHT
3	T	SRA (B,L)
3	F	SLA (B,L)
4	T	SRL (B,L)
4	F	SLL (B,L)
5	T	RR (B)
5	F	RL (B)
6	T	RRC (B)
6	F	RLC (B)

The procedure defines the fill depending on the operation as follows :

n	right	fill
1	T	MSB OF THE REGISTER
3	T	MSB OF THE REGISTER
5	T	LSR OF THE REGISTER
6	T	FLAG CARRY
5	F	MSB OF THE REGISTER
6	F	FLAG CARRY

The procedure also it sets the flags after the execution and during the operation checks for overflow condition except in the cases of :

- SRA (B,L) ,SDL (B,L) ,SKI (B,L) because for the operation SDL and SKI the flag PV is undefined and for the operation SRA the PV flag must be cleared.

The C flag is set depending on the MSB of the register except in the case of RLB or RR when the C flag is set depending on the last rotated bit.

The Z flag is set if the result is equal to zero.

#### PARAMETERS

----- : as have been described above.

#### GLOBAL VARIABLES

- : - 'r' is the used register.
- dst is the number of the destination register.  
Range 0..15.

#### LOCAL VARIABLES

- : - save saves the original number of the register in the case that the operation is executed for double word registers.
- index is a copy of the dst variable in integer value.Range 0..15.
- j is used as an index in the loop that denotes how many times the operation will be executed.Range 1..8 or 1..16 or 1..32 depending on the type of the register.
- k is used as an index to the loop that denotes in how many registers the operation will be executed.Range 1..2.

```

721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761

```

- m is used as an index to the loop that denotes in how many bits the operation will be executed, depending on the length of the register.Range 7 or 15.
- fill is the type of fill'.Range T or F.
- check is used to check if overflow occurs.Range T or F.
- temp,temp1 are used to interchange the bits in the rotations.Range T or F.

The procedure calls : The procedures upperlow, setindex.  
The procedure is called by : ROTATFSHIF.

\*\*\*\*\*)

```

procedure shiftrightleft( right: boolean; n: integer);

```

```

var
    save, index, j, k, m: integer;
    fill, check, zero, temp, temp1: boolean;

```

```

begin (al*)
    fill := false;(*fill is true or false depending on the *)
        (* function *)
    rlfcl(pv):= false; (*clear the overflow*)
    index := dst;(*dst register*)
    upperlow(index, length, upper, low);
    setindex(length, index);(*return values of correct *)
    (* register's index *)
    save := index;(*saves the original value of the register 's *)

```

```

784 (* index *)
785 if ((n = 3) or (n = 1)) and right then
786   (*SRA (B,L) or SDA (R,L)*)
787   fill := r[index]upperl;(*msh of the register*)
788   check := r[index]upperl;(*save the msh to check for *)
789   (* overflow *)
790
791 if right then begin (*2*)
792   for j := 1 to times do begin(*3*)(*0..32 times depending *)
793     (*on the length of the register and the type of the function*)
794     if n = 5 then
795       fill := r[index]flowl;(*RK*)
796     if n = 6 then
797       fill := r[fcw]fc1;(*RRC*)
798     temp := fill;(*temp is used in the loop to interchange *)
799     (* the bits *)
800     index := save; (*in case when it will be executed *)
801     (*twice for double word registers*)
802     for k := 1 to length do begin(*double registers only *)
803       (* have value of length = 2*)
804       for m := upper downto low do begin (*5*)
805         temp1 := r[index]m;(*saves the original value of *)
806         (* the bit *)
807         zero := zero or temp;(*check for zero*)
808         r[index]m1 := temp;(*sets the bit to the new value*)
809         temp := temp1(*sets temp to the original value *)
810         (* of the bit *)
811       end; (*5*)
812       if (length = 2) and (k = 1) then (*double word register*)
813         index := index + 1
814         (*and it continues to the next register*)
815       end; (*4*)
816       if n = 6 then (*RRC used if it is double register only*)
817         r[fcw]fc1 := temp; (* saves the last bit *)
818       if (n <> 2) and (n <> 4) and not r[fcw]low then
819         (* SRL SDL *)
820
821
822
823
824
825
826
827

```

```

841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881

    r[fcw]pv] := check <> r[index]upper];
end; (*3*)
if n = 5 then (*RK*)
    r[fcw]lc] := r[index]upper] (* sets the carry flag *)
else
    (*in all the other cases*)
    r[fcw]lc] := temp (* the last transferred bit*)
end else begin (*2*)(*14*)
    for j := 1 to times do begin(*15*)(*0..32 times depending *)
        (*on the size of the register or the type of the function*)
        if n = 6 then
            fill := r[fcw]lc];(*RL(C)*)
        (* original fill equals to the existed carry *)
        if n = 5 then
            fill := r[index]upper];(*RL*)
        (* else equals to the value of the MSB *)

        temp := fill;(*temp is used to interchange the bits *)
        (* in the loop *)
        index := save; (* in case when it will be executed*)
        (*twice for double registers*)
        if length = 2 then
            index := index + 1;(*if double register the loop starts *)
        (* from the lower register *)
        for k := 1 to length do begin (*16*)
            for m := low to upper do begin (*17*)
                temp1 := r[index]lm];(*saves the original value *)
                (* of the bit *)
                zero := zero or temp;(*check for zero*)
                r[index]fm] := temp;(*set the bit to new value*)
                temp := temp1;(*set the temp to the old value *)
                (* of the bit *)
            end; (*17*)
        if (length = 2) and (k = 1) then
            index := index - 1 (* double word register *)
        end; (*16*)
    end;
end;

```

```

904 if n = 6 then (*it is used only if the register is double *)
905     (* in the function RPLC *)
906     rfcw[cl] := temp;
907     if (n <> 2) and (n <> 4) and not rfcw[lpv] then
908         rfcw[fpv] := check <> rindex[upper];
909     end; (*15*)
910     if n <> 5 then
911         rfcw[cl] := temp(*all the other cases*)
912     else
913         rfcw[cl] := rindex[low] (*in the case of RL*)
914     end; (*14*)
915     rfcw[fs] := rindex[upper];(*set the sign flag*)
916     rfcw[fz] := not zero;(*set the zero flag*)
917     if (n = 3) and right then (*SRA clear the overflow*)
918         rfcw[fpv] := false
919     end; (*1 shift right*)
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947

```

```

927 begin (*1*)
928     setsrcdst(dst, select);(*select the type of the operation*)
929     setlength1(length, 1);
930     case opcode of
931         188, 190:
932             (* these opcodes are used for decimal multiplication *)
933             (* and division *)
934             begin (*RRD(R) or RLD(R) (*2*)
935                 retvalreq(length, dst, value);(*value ← R(R) dst*)
936                 retvalreq(length, select, val);(*val ← R(R) select*)
937                 case opcode of
938                     188:
939                         begin (*multiplication*)(*RRD(R) (*3*)
940
941
942
943
944
945
946
947

```



```

961 valuebytes(value, val);(*returns the value *)
962 (* of each byte *)
963 val := lsbsrc;(*dst 0:3 ← src 0:3*)
964 value := lsbdst * 16;(*src 4:7 ← dst 0:3*)
965 value := msbsrc + value;(*src 0:3 ← src 4:7*)
966 rlfcl[2] := value = 0
967 (* sets the zero flag *)
968 end; (*3*)
969
970 190: begin (*division*)(*RLDB*)(*4*)
971 valuebytes(val, value);(*src,dst*)
972 value := lsbdst;(*src 0:3 ← dst 0:3*)
973 value := value + 16 * lsbsrc;(*src 4:7 ← src 0:3*)
974 val := 16 * msbdst + msbsrc;(*dst 0:3 ← src 4:7*)
975 rlfcl[2] := val = 0
976 (* sets the zero flag *)
977 end (*4*)
978
979 end; (*case opcode 188,190*)
980 setreg(length, value, dst);(*set the register to *)
981 (* the new value *)
982 setreg(length, val, select);
983 ic := ic + 1
984 end; (*2*)
985 178, 179:
986 (* all the types of rotate operations and shift ones *)
987 begin (*5*)
988   ic := ic + 1;
989   if (select = 15) or (select = 7) or (select = 13)
990   or (select = 5) then
991     length := 2;(*double register*)
992     if (select = 10) or (select = 2) or (select = 14)
993     or (select = 6) then
994       times := 2;(*twice the operation will be executed*)
995     else
996       times := 1;(*once the operation will be executed*)
997
998 1000
999 1001

```

```

1024 case select of
1025 8, 10: (*RL(R) if R then once else twice*)
1026   shiftrightleft( false, 5);
1027 0, 2: (*RLC(B) if 0 then once else twice*)
1028   shiftrightleft( false, 6);
1029 12, 14: (*RR(R) if 12 then once else twice*)
1030   shiftrightleft( true, 5);
1031 4, 6: (*RRC(R) if 4 then once else twice*)
1032   shiftrightleft( true, 6);
1033 11, 3, 9, 1, 15, 7, 13,
1034 5:
1035   (* SDA (R,L) ,SDL(R,L) SIA (R,L) SLL (R,L) *)
1036   begin (*6*)
1037     correct := true;
1038     mapic, 1, 4, temp):(*temp ← src*)
1039     ic := ic + 2;
1040     if (select = 3) or (select = 11) or (select = 15)
1041     or (select = 7) then begin (*6a*)(*the number of the *)
1042       (* register *)
1043       if temp < 15 then begin (* if SDD or SDL then the *)
1044         (* source is R *)
1045         retvalreq(1, trunc(temp), val);(*val ← [R]*)
1046         temp := val
1047       end else
1048         correct := false;
1049       end; (*6a*)
1050     if correct then begin
1051       if temp >= 32768 then begin (*7*)
1052         negative := true;(*so right*)
1053         times := 65536 - trunc(temp)
1054       end else begin (*7a*) (*7a*)
1055         negative := false;(*so left*)
1056         times := trunc(temp)
1057       end; (*7a*)
1058     if length = 2 then

```

```

1081      correct := times < 33(*double register so *)
1082              (*maximum permitted is 32 *)
1083
1084      else if length = 1 then
1085      correct := times < 17 (*normal register so *)
1086              (* maximum permitted is 16 *)
1087
1088      else
1089      correct := times < 9;(*short register so *)
1090              (* maximum permitted is 8 *)
1091
1092      end;
1093      if correct then
1094      case select of
1095      11, 15:
1096          begin (*8*)(*SDA B,L*)
1097              right := negative; (* if negative number *)
1098              shiftrightleft( right, 1)
1099              end; (*8*)
1100
1101      (*if left then the fill is false else the subroutine finds the fill*)
1102      3, 7:
1103          begin(*9*)(*SDL (B,L) logical (RIGHT or LEFT)*)
1104              right := negative;
1105              shiftrightleft( right, 2)
1106              end; (*9*)
1107
1108      9, 13:
1109          begin(*10*)(*SLA (b,L) or SKA (B,L)*)
1110              (*shift left arithmetic or right*)
1111              right := negative;
1112              shiftrightleft( right, 3)
1113              end; (*10*)
1114
1115      1, 5:
1116          begin(*11*)(*SLL logical or SRL*)
1117              right := negative;
1118              shiftrightleft( right, 4)
1119              end (* 11 *)
1120          end (* case *)
1121      end (* 6 *)

```

```
1144         end (* case select *)
1145     end (* 5 *)
1146 end (*case select opcode*)
1147 end (*1 KOTALE SHIFT*)
1148 end;
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
```

1201  
1202  
1203  
1204  
1205 (\*\*\*\*\*  
1206  
1207  
1208

PROCEDURE BLOCKTRANSFER

1209 FUNCTION  
1210 ----- The procedure executes all the oncodes of type CPU and LDD  
1211 as follows :  
1212

1213                    OPCODE = 186 OR 187 (byte operation or word one)

8th half byte of the opcode	4 th half byte of the opcode	type of operation	dst	src	r
-----	-----	-----	-----	-----	-----
-	B	CPD (B)	R	IR	R
-	C	CPDR (B)	R	IR	R
-	0	CPI (B)	R	IR	R
-	4	CPIR (R)	R	IR	R
-	A	CPSD (R)	IR	IR	R
-	E	CPSDR (B)	IR	IR	R
-	2	CPSI (R)	IR	IR	R
-	6	CPSIR (B)	IR	IR	R
R	9	LDD (B)	IR	IR	R
0	9	LDPR (B)	IR	IR	R
R	1	LDI (B)	IR	IR	R
0	1	LDIR (B)	IR	IR	R

1232  
1233 The procedure finds the proper function and then it calls  
1234 the nested procedure compare to execute the function.

1235  
1236 PARAMIFRS  
1237 ----- : - None.

1238  
1239 GLOBAL VARIABLES

1240

```

1264 : - ic is the decimal value of the counter.
1265 Range 0 ..maxmem.
1266
1267
1268 LOCAL VARIABLES
1269 : - select is the value of the fourth half
1270 byte denoting the type of the operation.
1271 Range is 0..15.
1272 - j is used as an index in a for loop.
1273 - length denotes the length of the register.
1274 Range 0..1.
1275 - l denotes the corresponding length in half
1276 bytes in the case to reference to the memory.
1277 Range 2 or 4.
1278 - src,dst are the source and destination operands
1279 of the opcode.Range 0..15.
1280 - rcounter is the register that it is used to de-
1281 note how many times the operation will be execu-
1282 ted in the case of the repeat modes.the procedu-
1283 re does not check for the size of the contents of
1284 the register because this it is checked by
1285 the procedures map and setmem.
1286 - incdec denotes the increment or decrement step
1287 for the source operand or the source and the
1288 destination operand depending on the type of
1289 the operation.Range is +- 1 or +- 2.
1290 - repetition is the value of the contents of
1291 the counter register that denotes how many
1292 times the operation will be executed.Range
1293 1..maxmem.
1294 - cccode is the test condition which is speci-
1295 fied in the eight half byte of the opcode
1296 and if it is satisfied then terminates the
1297 operation.Range 0..15.
1298 - times is a copy of the variable repetition
1299 but it is decremented in each execution and
1300
1301
1302
1303
1304
1305
1306
1307

```



```

1321         final if it becomes equal to zero or the test
1322         condition is fulfilled sets the counter regi-
1323         ster to his new value.
1324
1325         - answer is the returned value from the called
1326         procedure testcc that denotes if the condition
1327         has been satisfied.Range T or F.
1328
1329         - temp is the decimal value of the source operand
1330         which is always IR.
1331
1332         - x,val are temporary values used in the mapping
1333         and setting operations.
1334
1335         - The procedure first decodes all the the parameters of the opcode
1336         as dst,src,rcounter and test condition.
1337
1338         - Assumes as default that the operation is increment and it changes
1339         the increment step to negative number if the operation is decre-
1340         ment.
1341
1342         - Finds if the dst operand is IR or K depending on the type of the
1343         operation.
1344
1345         - Finds the decimal value of the source.
1346
1347         - In the case of CP type operation calls the nested procedure compare
1348         with parameter the times that it will be executed.The procedure com-
1349         pare enters a while loop and it uses the temporary registers R17 (dst)
1350         and R18 (src) to make the comparisons without to affect the original
1351         values until the test condition is satisfied or the specified
1352         times is executed.
1353
1354         - In the case of LD operation the procedure retrieves the value of
1355         the source (memory) and sets the destination memory to the same
1356         value for the specified times.
1357
1358         - If the counter register becomes zero then it sets the PV flag.
1359
1360         - If the result of the compare is zero sets the Z flag.
1361
1362         The procedure is called by :EXECUTE.
1363         The procedure calls :The procedures setsrcdst,setlength,map
1364         retvalreg,setmem,setreq.
1365
1366         *****)

```



```
1384  
1385  
1386  
1387  
1388  
1389  
1390  
1391  
1392  
1393  
1394  
1395  
1396  
1397  
1398  
1399  
1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1420  
1421  
1422  
1423  
1424  
1425  
1426  
1427  
  
procedure BLUOTRANSFER;  
var  
  select, j, length, l, src, dst, rcouter, incdec,  
  repetition : integer;  
  value, val, temp, cccode, x, times: real;  
  answer: boolean;
```

```

1441
1442
1443
1444
1445  (*****
1446
1447
1448
1449
1450  FUNCTION
1451  ----- : As above has been defined.
1452
1453  PARAMETER
1454  ----- : - repetition is the times that the compare operation
1455           will be executed.
1456
1457  GLOBAL VARIABLES
1458  ----- : - value,temp,cccode,times x,are local the procedure
1459           BLOCKTRANSFER.
1460
1461  LOCAL VARIABLES
1462  ----- : - j is used as an index into the while loop.
1463
1464  The procedure calls : The procedures onetwocompl,ADD, testcc,map.
1465  The procedure is called by :BLOCKTRANSFER.
1466
1467  *****
1468
1469
1470
1471  procedure compare(repetition: integer);
1472
1473
1474  var
1475      j: integer;
1476
1477  begin (*1*)
1478      j := 1;
1479      while not answer and (j <= repetition) do begin (*2*)
1480
1481

```

```

1504 setreq(1, value, temporary);(*Rtemporary ← dst*)
1505 setreq(1, temp, temporary + 1); (*Ktemporary+1 ← src*)
1506 onetwocompl(temporary + 1, 1, 2, false);
1507 (* Rtemporary +1 ← - Rtemporary +1 *)
1508 ADD(temporary, temporary + 1, 1, true, false); (*subtract*)
1509 testcc(answer, cccode);(*test if the termination condition *)
1510 (* is satisfied *)
1511 j := j + 1;(*increment the index*)
1512 times := times - 1;(*decrement the value of the rcounter*)
1513 x := x + incdec; (*increment or decrement 1 or 2*)
1514 map(x, 1, 1, temp);(*temp ← mem[src]*)
1515 case select of
1516   0, 4, 8, 12:
1517     null;(*dst is R*)
1518   2, 6, 10, 14:
1519     begin (*3*)(*dst is IR*)
1520       val := val + incdec;(*increment or decrement the *)
1521       (* dst 1 or 2 *)
1522       map(val, 1, 1, value);
1523     end (* 3 *)
1524   end (* case *)
1525 end;
1526 (*while*)
1527 rftcw[z] := answer (*if the answer is true then sets *)
1528 (* the zero flag *)
1529 end; (* compare*)

```

```

.
1535 begin (*1*)
1536   answer := false; (* default value is false *)
1537   setsrcdst(src, select);(*src is always IR,select is the type*)
1538   (*of the operation*)
1539   setlength(length, 1);

```

```

1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601

if length = 0 then
    incdec := 1 (*one byte*)
else
    incdec := 2;(*one word*)
    ic := ic + 1;
    map(ic, 1, 1, x);(*x + mem[ic] = rcounter*)
    (* is the used register as counter *)
    rcounter := trunc(x);
    ic := ic + 1;
    map(ic, 0, 1, x);(*low byte*)
    dst := trunc(x);(*dst is either IR or R*)
    (* depending on the type of the operation *)
    map(ic, 1, 1, cccode); (* finds the test condition *)
    ic := ic + 1;
case select of
    0, 9, 10, 12, 14:
        incdec := -incdec;(*decrement*)
    0, 1, 2, 4, 6:
        null (* increment *)
    (*case select*)
end;
case select of
    0, 4, 8, 12:
        retvalreg(length, dst, value);(*value + [rkdst]*)
    1, 2, 6, 9, 10, 14:
        begin (*2*)
            retvalreg(1, dst, val);(*dst is IR*)
            map(val, 1, 1, value) (*value + mem[ Rdst]*)
        end (* 2 *)
    end; (*case*)
    retvalreg(1, src, x);(*x + [Ksrc]*)
    map(x, 1, 1, temp);(*temp + mem[Ksrc]*)
    retvalreg(1, rcounter, times); (*times + (rcounter)*)
case select of
    0, 4, 8, 12:
        begin (*10*)

```

```

1624   if (select = 8) or (select = 0) then
1625     repetition := 1(*CPD(R) or CPI(R)**)
1626   else
1627     repetition := trunc(times);(*CPDR(B) or (CPIR(B)**)
1628     compare(repetition)
1629   end; (*10*)
1630   2, 6, 10, 14:
1631   begin (*11*)
1632     if (select = 10) or (select = 2) then
1633       repetition := 1(*CPSD(B) or CPSI(R)**)
1634     else
1635       repetition := trunc(times);(*CPSDR(R) or (PSIR(B)**)
1636       compare(repetition)
1637     end; (*11*)
1638   1, 9:
1639   begin (*12*)
1640     if ccode = 8 then
1641       repetition := 1(*LDD(H) or LDI (B)*)
1642     else
1643       repetition := trunc(times);(*LDDR(B) or LDIR(B)**)
1644       for j := 1 to repetition do begin (*13*)
1645         setmem(val, 1, 1, temp);(*mem[dst] ← mem[src]**)
1646         x := x + incdec;(*src ← src + or - 1 or 2*)
1647         val := val + incdec;(*dst ← dst + or - 1 or 2*)
1648         map(x, 1, 1, temp); (*temp ← new value of mem[src]**)
1649         times := times - 1(*rcounter ← rcounter - 1*)
1650       end (*13*)
1651     end (*12*)
1652   end ; (*case*)
1653   setreq(1, times, rcounter); (* sets the rcounter to the new value *)
1654   rffcw[low] := times = 0(*sets the pv flag if times = 0*)
1655   end; (*) RLOCTransFER*)
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667

```

1681

1682

1683

1684

1685

1686

1687

1688

1689

1690

# FUNCTION

----- The procedure executes all the opcodes of translate and test type as follows:

fourth half byte of the opcode	eight half byte of the opcode	Base opcode 184 operation	dst	src	r
0	0	TRDR	IR	IR	R
C	0	TRDRB	IR	IR	R
0	0	TRIR	IR	IR	R
4	0	TRIRB	IR	IR	R
			src1	src2	
A	0	IRIDB	IR	IR	R
E	E	IRIDRB	IR	IR	R
2	0	IRIB	IR	IR	R
6	E	IRIRB	IR	IR	R

# PARAMETERS

----- :None.

# GLOBAL VARIABLES

----- : - ic is the current value of the decimal counter  
Range 0..maxmem.

# LOCAL VARIABLES

----- : - select is the value of the fourth half byte of  
the opcode defining the type of the operation.



- j is used as increment index into the while loop  
Range 1..times.
- rcounter is the register that denotes how many times  
the operation will be executed.Range 1..maxmem.
- times is the contents of the rcounter register.  
Range 0..maxmem.
- dst is the destination operand or the src1 operand  
depending on the type of the operation.Range 1..15.
- incdec has value +-1 depending if the operation  
is decrement or increment.
- repet is an integer copy of the variable times  
and it is used in the while loop.Range is 1 or  
1.. to the contents of the rcounter register.
- base, offset are the base address and the offset  
address.Base address is the contents of the src and  
offset is the contents of destination's memory.  
Range 0..maxmem.
- val is the contents of the destination register.  
Range 0..maxmem.
- value is the contents of the source register.Ran-  
ge is 0..maxmem.
- x is used as a temporary variable.

#### The procedure :

- decodes all the parts of the opcode.
- finds all the initial addresses of the destination and the source.
- finds if the operation is increment or decrement and also if  
the operation will be executed once or more times.
- In the case of TRDB,IRDRB,IPB,IRKB translates and sets the  
destination to the new value.
- In the case of translate and test it translates,it does not  
change the destination and terminates either if RHL becomes <0  
or it is executed the specified times.
- the Z flag is set if RHL becomes zero.
- The PV flag is set if the rcounter becomes zero.



```

1801
1802
1803
1804
1805
1806 The procedure calls : The procedures setsrcdst,retvalreq,map,setmem,
1807      setreq.
1808 The procedure is called by : FxFULF.
1809
1810
1811
1812 *****
1813 *****
1814 *****
1815 *****
1816 *****
1817 *****
1818 *****
1819 *****
1820 *****
1821 *****
1822 *****
1823 *****
1824 *****
1825 *****
1826 *****
1827 *****
1828 *****
1829 *****
1830 *****
1831 *****
1832 *****
1833 *****
1834 *****
1835 *****
1836 *****
1837 *****
1838 *****
1839 *****
1840 *****
1841 *****

procedure TRANSLATE;

var
select, j, times, dst, src2, incdec, repet, rcounter: integer;
offset, base, x, val, value, temp: real;

begin (*1*)
setsrcdst(dst, select); (*dst is the dst register, select *)
                          (* denotes the type of the operation*)
ic := ic + 1;
map(ic, 1, 1, x);
rcounter := trunc(x);(*rcounter denotes how many times*)
(* the operation will be executed *)
retvalreg(1, rcounter, x);(*x + (rcounter)**)
times := trunc(x);
ic := ic + 1;
map(ic, 0, 1, x);(*x + src2 or src depending on the function*)
src2 := trunc(x);
ic := ic + 1;
case select of
4, 6, 12, 14:
repet := times; (*1RDRB , 1RJRb, 1K1DRb, 1K1JRb*)
0, 2, 8, 10:
repet := 1

```

```

1864      (*IRDR , TRID, TRIDB, IPIIB*)
1865      end; (* case select *)
1866      case select of
1867      8, 12, 10, 14:
1868      incdec := -1;(*decrement*)
1869      0, 4, 2, 6:
1870      incdec := 1(*increment*)
1871      end;
1872      (*case*)
1873      retvalreg(1, dst, val);(*val + (Rdst)*)
1874      retvalreg(1, src2, value);(*value + (Rsrc2)*)
1875      case select of
1876      0, 4, 8, 12:
1877      begin (*2*)(*TRDB , IRDRB, TRID, TRIRB*)
1878      for j := 1 to repet do begin (*3*)
1879      base := value; (*Rsrc)*)
1880      map(val, 1, 2, offset); (*offset + mem(dst)*)
1881      base := base + offset;(*srcdst)*)
1882      map(base, 1, 2, tempo);(*temp + mem(src(dst))*)
1883      setmem(val, 1, 2, tempo); (*dst + mem(srcdst)*)
1884      times := times - 1;(*rcounter + rcounter = 1*)
1885      val := val + incdec (*dst + dst + or - 1*)
1886      end (*3*)
1887      end;
1888      10, 14, 2, 6:
1889      begin (*4*) (*all the translate and test modes*)
1890      j := 1;
1891      x := 0;(*hercomes one for the first loop*)
1892      while (j <= repet) and (x = 0) do begin (*5*)
1893      base := value; (*base + (Rsrc)*)
1894      map(val, 1, 2, offset); (*offset - mem(src)*)
1895      base := base + offset; (*base + src2(src)*)
1896      map(base, 1, 2, x); (*x+ mem (src2(src))*)
1897      setreg(0, x, 1); (*Rd1 + x*)
1898      val := val + incdec; (*src1 + src1 + or - 1*)
1899      times := times - 1;
1900
1901
1902
1903
1904
1905
1906
1907

```

```

1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961

      j := j + 1
      end; (*5*)
      r[fcw][z] := x = 0 (* set the zero flag if RH1 = 0*)
      end (*4*)
      end; (*case select*)
      r[fcw][pv] := times = 0; (*set the pv flag if rcounter = 0*)
      setreq(1, times, rcounter) (* sets the rcounter to the new value *)
      end; (*1 TRANSLATE*)

```

```

1984 (*****
1985 *****
1986 *****
1987 *****
1988 *****
1989 *****
1990 *****
1991 *****
1992 *****
1993 *****
1994 *****
1995 *****
1996 *****
1997 *****
1998 *****
1999 *****
2000 *****
2001 *****
2002 *****
2003 *****
2004 *****
2005 *****
2006 *****
2007 *****
2008 *****
2009 *****
2010 *****
2011 *****
2012 *****
2013 *****
2014 *****
2015 *****
2016 *****
2017 *****
2018 *****
2019 *****
2020 *****
2021 *****
2022 *****
2023 *****
2024 *****
2025 *****
2026 *****
2027 *****

PROCEDURE STACKINTERCHANGE

FUNCTION
----- The procedure interchanges the two copies of the R15
register when the simulator changes mode from system
to normal and from normal to system.
Also it sets the system flag depending on the value of
S/N flag.

PARAMETERS
----- : None.

GLOBAL VARIABLES
----- : - 'r' is the used registers. The used indexes are
constant declarations.
- system denotes if the mode of operation is system
or normal. Range 1 or F.

LOCAL VARIABLES
----- : None.

The procedure calls : None.
The procedure is called by : TRAP, IRET, LPDS, LDCIL, MONITOR.

*****

procedure STACKINTERCHANGE;
begin (* 1 *)
  r(temporary) := r(ospoff); (* saves the contents of R15 to the*)

```

```
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
```

```

(* register R17 *)
r(nspoff) := r(23);(* sets the R15 to the contents of the R23*)
(* which contains the system stackpointer *)
r(23) := r(temporary);(* saves the old R15 in the R23*)
system := r(fcwl[sn]);(* sets or resets the system flag*)
end; (* 1 STACKINTERCHANGE *)
```

```

2104  (*****
2105  (*****
2106  (*****
2107  (*****
2108  (*****
2109  (*****
2110  (*****
2111  (*****
2112  (*****
2113  (*****
2114  (*****
2115  (*****
2116  (*****
2117  (*****
2118  (*****
2119  (*****
2120  (*****
2121  (*****
2122  (*****
2123  (*****
2124  (*****
2125  (*****
2126  (*****
2127  (*****
2128  (*****
2129  (*****
2130  (*****
2131  (*****
2132  (*****
2133  (*****
2134  (*****
2135  (*****
2136  (*****
2137  (*****
2138  (*****
2139  (*****
2140  (*****
2141  (*****
2142  (*****
2143  (*****
2144  (*****
2145  (*****
2146  (*****
2147  (*****

PROCEDURE IRAP

FUNCTION
----- The system supports the following IRAP conditions :
- The user calls the system mode.
- The user tries to execute unimplemented instructions.
- The user tries to execute privilege instructions as
  EL1,PPS and the mode of operation is normal.
- The procedure saves the contents of the nspoff (R15) in
  the register k23 and sets the R15 with the value of the
  R23 containing the value of the system stackpointer.
- Push in the system stack by order the IC,FCW,and the
  current value of the opcode.
- Jumps in the PSA area and adds the base address of the
  PSA area which is containing in the psaoff register with
  the offset value of the trap which is passed as a param-
  eter.
-Copies the existed value in the memory in the IC and jumps
to the new address.
-Loads the new value of the FCW
- Increases the offset by 2 and copies the existed value

PARAMETERS
----- : - by value -'n' is the type of the IRAP condition.Range 0..30.
- opcode is the type of the opcode which is re-
  sponciple for the IRAP.

GLOBAL VARIABLES
----- : None.

```

```

2161
2162
2163
2164
2165 LOCAL VARIABLES
2166 ----- : - ssp , val are used for multipurposes as it is
2167           defined in the procedure.
2168 The procedure calls : The procedures retvalreg, setreg.
2169 The procedure is called by : LPDS, INPUTOUTPUT, EXECUTE , LDCIL,
2170                           STACKINTERCHANGE.
2171
2172
2173 *****
2174 *****
2175 *****
2176 *****
2177 *****
2178 *****
2179 *****
2180 *****
2181 *****
2182 *****
2183 *****
2184 *****
2185 *****
2186 *****
2187 *****
2188 *****
2189 *****
2190 *****
2191 *****
2192 *****
2193 *****
2194 *****
2195 *****
2196 *****
2197 *****
2198 *****
2199 *****
2200 *****

procedure IRAP(n , opcode : integer);

var
  ssp , val: real;

begin (*1*)
  (* INITIAL SITUATION
  -----
  ; empty ; ← STACKPOINTER
  -----*)

  STACKINTERCHANGE;
  retvalreg(1, nspoff, ssp); (* ssp ← the contents of the R15*)
  if ssp - 6 >= 0 then begin (*2*)
    setmem(ssp, 1, 4, ic); (* saves the ic in the system stack *)
    retvalreg(1, fcw, val); (* val ← the contents of the FCW register *)
    setmem(ssp-2, 1, 4, val); (* saves the FCW in the stack *)
    setmem(ssp-4, 1, 4, opcode); (* saves the opcode *)
    setreg(1, ssp-4, nspoff); (* sets the system stackpointer to *)
                          (* new value *)
    retvalreg(1, psao, val); (* val ← the base address of the PSA *)
                          (* area. *)
    val := val + n ;      (* n is the offset depending on the *)
  end;

```



```

2224
2225 (* the kind of the TRAP *)
2226 map(val,1,4,ssp);(* ssp ← is the new value of the FCW register.*)
2227 setren(1,ssp,fcw);(* sets the FCW register to the new value*)
2228 map(val+2,1,4,ssp);(*ssp ← is the new value of the ic*)
2229 ic := trunc (ssp);
2230 end else(*2*)
2231 error(2) (* underflow *)
2232 (* FINAL SITUATION
2233 -----
2234 | OP CODE | ← SIACPUINTER
2235 -----
2236 | FCW |
2237 -----
2238 | IC |
2239 -----*)
2240 end: (* 1 TRAP *)
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267

```

```
2281
2282
2283
2284
2285 (*****
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
```

```

*****
PROCEDURE LPDS

FUNCTION
-----
The contents of the source operand are loaded into the
program status (PS) , loading the FLAGS and control word
(FCW) and the program counter (PC).The new value of the
(FCW) does not becomes effective until the next instru-
ction.The instruction LPDS is a privileged instruction
The addressing modes are three :
opcode      src      type of address.
57          <> 0      IR
120         <> 0      X
120         = 0      DA

If the S/N flag changes to NORMAL mode then the procedure
calls the procedure STACKINTERCHANGE to interchange the
the copies of the R15.

GLOBAL VARIABLES
-----
: - ic is the decimal counter .Range is 0 .. maxmem.

LOCAL VARIABLES
-----
: - value,val are used as temporary variables
to save and send values or parameters to
other procedures.Range 0..maxinteq.

The procedure calls :The procedures TRAP,retvalreq,DaorX,map,setreg,
STACKINTERCHANGE.
The procedure is called by : FxECUIF.

*****
)
```

```

2344 procedure LPDS;
2345
2346
2347
2348
2349 var
2350 value, val: real;
2351
2352 begin (*1*)
2353 if opcode = 57 then begin (*2*)
2354     retvalreg(1, src, value);(*value ← (Rsrc)*
2355     ic := ic + 1
2356 end;
2357 if opcode = 121 then
2358     DAOrX(src, value);(*DA or X mode of address*)
2359     map(value, 1, 4, val);(*val ← mem(value)*
2360     setreg(1, val, fcw);(*rlfcw ← mem(value) new value of *)
2361     (* the FCW register *)
2362     map(value + 2, 1, 4, val);
2363     setreg(1, val, counter);(*PC register ← mem(value)*
2364     ic := trunc(val); (* changes the value of the ic *)
2365     if not rlfcw[fsn] then
2366         STACKINTERCHANGE;
2367     end; (*1 LPDS *)
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387

```

2401  
2402  
2403  
2404  
2405 (\*\*\*\*\*  
2406  
2407  
2408  
2409  
2410  
2411  
2412  
2413  
2414  
2415  
2416  
2417  
2418  
2419  
2420  
2421  
2422  
2423  
2424  
2425  
2426  
2427  
2428  
2429  
2430  
2431  
2432  
2433  
2434  
2435  
2436  
2437  
2438  
2439  
2440  
2441  
2442  
2443

\*\*\*\*\*  
PROCEDURE LDCIL  
FUNCTION  
----- The procedure executes all the opcodes of the LDCIL type.  
The opcodes have the same value 140 and they differ in the  
fourth half byte as follows :.  
fourth half  
byte  
----- operation  
9 LDCILB FLAGS,R  
1 LDCILB R,FLAGS  
A LDCIL FCW,R  
2 LDCIL R,FCW  
B LDCIL REFRESH,R  
3 LDCIL R,REFRESH  
C LDCIL PSAPSEF,R  
8 LDCIL R,PSAPSEG  
D LDCIL PSAPOFF,R  
5 LDCIL R,PSAPOFF  
E LDCIL NSPSEG,R  
6 LDCIL R,NSPSEG  
F LDCIL NSPOFF,R  
7 LDCIL R,NSPOFF  
The above opcodes are privileged and the procedure is called only  
in the system mode.  
The nested procedure exchange is used to execute the function.  
If the S/N flag change to normal mode due to the function  
LDCIL FCW,R then the procedure calls the procedure STACKINTERCHANGE  
to interchange the two copies of the R15.  
PARAMETERS  
-----: None

```

2464
2465
2466 GLOBAL VARIABLES
2467 -----: -ic is the current value of the counter.
2468
2469 LOCAL VARIABLES
2470 -----: -dst is the destination register.Range 0..23.
2471 -select is the value of the fourth half byte
2472 of the opcode denoting the type of the operation.
2473 Range 0..F.
2474 - val is used if the whole values of the registers
2475 will be exchanged as in the case of LDCIL R,FCW.
2476 Range 0..maxinteg.
2477
2478 The procedure calls : the procedures setsrcdst,exchange,retvalreq
2479 setreq.
2480 The procedure is called by : EXFCUIF.
2481
2482
2483 *****
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507

```

procedure LDCIL;

var  
dst, select: integer;  
val: real;

\*\*\*\*\*

```

2521
2522
2523
2524
2525  (*****
2526
2527
2528
2529
2530  PARAMETERS
2531  ----- : -dst,src are the destination and the source registers
2532           Range 0..23.
2533           - low ,upper denote the range of the bits that will
2534             be trasferred.Range 0..15.
2535
2536
2537
2538  *****
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562

```

PROCEDURE exchange.

```

procedure exchange(dst, src, low, upper: integer);

var
  j: integer;

begin (*1*)
  for j := low to upper do
    r[dst][j] := r[src][j](*copy from one reg to another *)
  end; (*1*)

begin (*1*)
  setsrcdst(dst, select);(*dst is either dst register or src register*)
  (*select denotes the type of the operation*)
  ic := ic + 1;
  case select of

```

```

2584 2:
2585   begin (*2*)(*LDCIL R, FCW*)
2586   retvalreg(1, fcw, val); (*val ← R[fcw]*)
2587   setreq(1, val, dst)
2588   end; (*2*)
2589   (*dst (2:7) ← FCW (2:7)
2590   dst (11:15) ← FCW (11:15)
2591   dst(0) ← 0
2592   dst (8:10) ← 0*)
2593 10:
2594   begin (*3*)(*LDCIL FCW, R*)
2595   exchange(fcw, dst, 2, /);
2596   (*FCW (2:7) ← src (2:7)*)
2597   exchange(fcw, dst, 11, 15);
2598   (*FCW (11:15) ← src (11:15)*)
2599   if not r[fcw][sn] then
2600     STACKINIFRCHANGE;
2601   end; (*3*)
2602 11:
2603   (*LDCIL RFFRESH, R*)
2604   exchange(refresh, dst, 1, 15);
2605   (*RFFRESH (1:15) ← src (1:15)*)
2606 5:
2607   begin (*4*)(*LDCIL R, RFFRESH*)
2608   exchange(dst, refresh, 1, 8);
2609   (*dst (1:8) ← refresh (1:8)*)
2610   r[dst][0] := false
2611   (*dst(9:15) ← undefined*)
2612   end; (*4*)(*dst [0] ← 0*)
2613 12:
2614   (*LDCIL PSAPSEF, R*)
2615   exchange(opause, dst, 8, 14);
2616   (*PSAPSEF (8:14) ← src (8:14)*)
2617 4:
2618   begin (*5*)(*LDCIL R, PSAPSEF*)

```



```

2641      retvalreq(1, psapseq, val);
2642      setreq(1, val, dst)
2643      (*dst (8:14) ← PSAPSEG (d: 14)
2644      dst (0:7) ← 0
2645      dst (15) ← 0*)
2646      end; (*5*)
2647      13:
2648      (*LDCIL PSAPUFF, R*)
2649      exchange(psaoﬀ, dst, 8, 15);
2650      (*PSAOFF (8:15) ← src (8:15)*)
2651      5:
2652      begin (*6*) (* LDCIL R, PSAOFF *)
2653          setreq(1, 0, dst);(*clear the Rdst*)
2654          exchange(dst, psaoﬀ, 8, 15)
2655          (*dst (8:15) ← PSAUFF (8:15)
2656          dst (0:7) ← 0*)
2657          end; (*6*)
2658      14:
2659      (*LDCIL R, NSPSEG *)
2660      exchange(nspseq, dst, 0, 15);
2661      (*NSPSEG ← src*)
2662      6:
2663      (*LDCIL R, NSPSEG *)
2664      exchange(dst, nspseq, 0, 15);
2665      (*dst ← NSPSEG*)
2666      15:
2667      (*LDCIL NSPOFF, R*)
2668      exchange(nspoff, dst, 0, 15);
2669      (*NSPOFF ← src*)
2670      7:
2671      (*LDCIL R, NSPUFF*)
2672      exchange(dst, nspoff, 0, 15)
2673      (*dst ← nspoff*)
2674      end; (*case*)
2675      end; (*1 LDCIL*)
2676
2677
2678
2679
2680
2681
2682

```

[illegible]

## PROCEDURE

# FUNCTION

This instruction is used to return to a previously executing procedure at the end of a procedure entered by trap ( as System Call instruction ).First, the identifier word associated with the trap is popped from the system processor stack and discarded.Then the contents of the location addressed by the system processor stack pointer are popped into the program status ,loading the flags and control word and the program counter.The new value of the PCW is not effective until the next instruction execution is completed.The next instruction executed is that addressed by the new contents of the PC.

The opcode is privileged one and it is executed only in the system mode.The procedure also checks for stack overflow and at the end it executes stack interchange

## PARAMETERS

None.

## GLOBAL VARIABLES

```
-----: - ic is the current value of the decimal counter
          the trap handler changes his value to the ad-
```

```

61
62
63
64
65
66
67 LOCAL VARIABLES
68 ----- : - val is the initial value of the stackpointer.
69           Range 0..maxmem.
70           - temp is used to map the stored values in the
71             stack to decimal values from hex ones.Range
72             0..maxinteq.
73
74 The procedure calls : the procedures retvalreg,map,setq,
75 STACKIN\FRCHANGE,error.
76 The procedure is called by : EXECUTE.
77
78 *****
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
27
```

```

124  man(val+2,1,4,temp);(*copies the saved FCw *)
125  setreg(1,temp,fcw); (*sets the FCw register to the *)
126  (* old value *)
127  man(val+4,1,4,temp);(* temp + the saved value of ic*)
128  ic := trunc(temp);
129  setreg(1,val+4,nspoff);(* increments the stackpointer*)
130  (* and sets the register*)
131  STACKINTERCHANGE;
132  end else(*2*)
133  error(?);
134
135
136  end;(* 1 IRET *)
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167

```

181  
182  
183  
184  
185 (\*\*\*\*\*  
186  
187  
188  
189  
190

PROCEDURE INPUTOUTPUT.  
  
-----  
The procedure executes all the input , output to and from  
registers or memory. The opcodes are privileged ones and  
the procedure is called only in the system mode. The base  
opcodes are nearly the same and they differ in the 4th  
and 8th half byte as follows:

operation	1st half byte	2nd half byte	3rd half byte	4th half byte	6th half byte	7th half byte	8th half byte
-----	-----	-----	-----	-----	-----	-----	-----
INR	60						
IN	61						
INP	58						
IN	59						
INDB	58						
IND	59						
INDRR	58						
INDR	59						
INIB	58						
INI	59						
INIRR	58						
INIR	59						
OUTR	58						
OUT	59						
OUTIR	58						
OUTI	59						
OUTDR	58						







```

364 (*****
365
366
367
368 PROCEDURE message.
369
370 FUNCTION
371 ----- The procedure reads characters from the screen and trans-
372 forms them to decimal values. The number of characters is
373 specified by the parameter 'n'.
374
375 PARAMETER
376 ----- : by reference - number is the returned equivalent decimal
377 value of the character or characters.
378 Range 0..65536.
379
380 by value - n is the number of the characters to
381 be read from the screen .Range 0..1.
382
383 GLOBAL VARIABLES
384 ----- : None.
385
386 LOCAL VARIABLES
387 ----- :
388 - ch is used to read characters from the screen.
389 Range ASCII characters.
390 - j is used as index in the for loop that denotes
391 the times of the repeat function. Range 0..n.
392 - k is the decimal value of each character. Range
393 0..127.
394
395 The procedure is called by : INPUTOUTPUT.
396 The procedure calls : None.
397
398 *****
399
400 procedure message(n: integer; var number: integer);
401
402 var
403
404
405
406
407

```

```

421         j, k: integer;
422         ch: char;
423
424     begin (* 1 *)
425         number := 0;
426         for j := 1 to n do begin(*2*)
427             if not eoln then
428                 read(ch)
429             else begin (* 3 *)
430                 ch := chr(13);(*carriage feed*)
431                 readln; (* skips a line *)
432                 end; (* 3 *)
433             k := ord(ch); (* transforms the character to decimal value *)
434             number := number * 256 + k
435             (* modulo 256 one byte *)
436             end; (* 2 *)
437         end;(* 1 message *)
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461

```

```

484 (*****
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527

```

PROCEDURE common.

```

----- The procedure assumes that the operation is repeat mode
with increment of the original address. It checks if the
operation is decrement and changes the sign of the step.
Also it sets the 'r' register of the opcode to his final
value and sets the overflow flag in the case that the 'r'
register becomes equal to zero.

PARAMETERS
----- : None

GLOBAL VARIABLES
----- : ( ALL LOCAL TO THE CALLING PROCEDURE INPUTOUTPUT )
- length denotes length of the register.Range
  0..1.
- four is the value of the 4th half byte of the
  opcode.Range 0..15.
- value is the contents of the 'r' register. The
  calling procedure sets the variable to zero if
  the mode is repeat one.Range 0..contents of
  'r' register.
- six is the number of the 'r' register.Range
  0..15.

LOCAL VARIABLES
----- : None.

The procedure is called by : INPUTOUTPUT.
The procedure calls : the procedure setereq.

*****

```

```
541
542
543
544
545
546
547 procedure common;
548
549 begin (* 1 *)
550   length := 1 div 2;(*increment*)
551   if (four = 8) or (four = 10) then
552     length := -length(*decrement*);
553   setreg(1, value, six);(*is set to zero or -1 *)
554   (* the original value *)
555   rlfcl[ov] := value = 0 (* set the overflow flag if *)
556                       (* the value = 0 *)
557   end;(* 1 common *)
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
```

```

604 *****
605 (*****
606 *****
607 *****
608 *****
609 *****
610 *****
611 *****
612 *****
613 *****
614 *****
615 *****
616 *****
617 *****
618 *****
619 *****
620 *****
621 *****
622 *****
623 *****
624 *****
625 *****
626 *****
627 *****
628 *****
629 *****
630 *****
631 *****
632 *****
633 *****
634 *****
635 *****
636 *****
637 *****
638 *****
639 *****
640 *****
641 *****
642 *****
643 *****
644 *****
645 *****
646 *****
647 *****

        PROCEDURE outputreq.

FUNCTION
-----
The procedure prints the contents of a byte or one word
register to the screen. It retrieves the contents of the
register and in the case of one word register it divides
the original value by 256 and then it prints the character
transforming the decimal value to equivalent character
using the built in function chr.

PARAMETER
-----
: by value - index is the number of the register. Range
0..15.

GLOBAL VARIABLES
-----
: - val is local to the calling procedure and it is
used to retrieve the contents of the register.
Range 0..maxinteq.

LOCAL VARIABLES
-----
: - val1 is the value of the left byte of the regi-
ster. Range 0..256.
- val2 is the value of the right byte of the regi-
ster. Range 0..256.

The procedure is called by : INPUTOUTPUT.
The procedure calls : the procedure retvalreq.

*****
procedure outputreq(index: integer);

```

```

661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702

var    val1, val2: integer;

begin(*1*)
    retvalren(length, index, val);
    val1 := trunc(val);
    val2 := val1 mod 256;(*second character*)
    val1 := val1 div 256;(*first character if it exists*)
    if length = 1 then
        write(chr(val1 mod 128));(*the first character is printed *)
        (* first *)
    write(chr( val2 mod 128 ))
end;(*1 outputreq*)

```



```

724 (*****
725 *****
726 *****
727 *****
728 *****
729 *****
730 *****
731 *****
732 *****
733 *****
734 *****
735 *****
736 *****
737 *****
738 *****
739 *****
740 *****
741 *****
742 *****
743 *****
744 *****
745 *****
746 *****
747 *****
748 *****
749 *****
750 *****
751 *****
752 *****
753 *****
754 *****
755 *****
756 *****
757 *****
758 *****
759 *****
760 *****
761 *****
762 *****
763 *****
764 *****
765 *****
766 *****
767 *****

```

```

PROCEDURE inputreg.

FUNCTION
----- The procedure calls the procedure message to read one
or two characters from the screen and then sets the
destination register.

PARAMETERS
----- : by value - index is the number of the destination
register. Range 0..15.

GLOBAL VARIABLES
----- : ( LOCAL TO THE CALLING PROCEDURE )
- temp is the returned decimal value equivalent
to the read characters.
- 1 denotes how many half bytes must be read.
Two half bytes are equivalent to one character.
Range 2 or 4.

The procedure is called by: INPUTUIPUT.
The procedure calls : The procedures message, setreg.
*****
procedure inputreg(index: integer);
begin(*1*)
    message(1 div 2, temp);
    setreg(lenoth, temp, index)
end;(*1 inputreg*)

```

```

781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822

```

```

begin(*1*)
three := src; (* src has the value of the third half byte *)
bytes := 0; (* one byte at least *)
four := dst; (* dst has the value of the fourth half byte *)
times := 1; (* at least one time will be executed any input *)
(*or output command*)
setlength1(length, 1); (* sets the variables length and 1 *)
ic := ic + 1;
case opcode of
    60, 61:
        (*IN (B) to registers*)
        inputreg(four);
        (*input byte or word to register No [four]*)
    62, 63:
        (*OUT (B) from the register*)
        outputreg(four);
        (*output byte or word from register No [four]*)
    58, 59:
        begin(*IN (B) or OUT (B) from ,to the memory*)(*4*)
            setsrddst(eight, six);(*eight is used as dummy variable *)
            (* because the value of the 5th half byte is always zero*)
            ic := ic + 1;
            setsrddst(seven, eight);
            ic := ic + 1;
            retvalreg(1, six, value);(*in case of repeat the value *)
            (*of the register specifies the times of execution*)
            if eight = 0 then begin(*5*)
                times := trunc(value);(*repeat until 'r' becomes zero*)
                value := 0
            end else(*5*)
                value := value - 1;(*once*)
            case four of
                5, 9, 1, 7, 11, 3:
                    error(6);
            (*SIN (B), SIND (R), SINDR (B), SINI (B), SINIR (B)*)

```

844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887

```

4: inputreg (three); (* input to register No [three]*)
6: outputreq(three);(*output from register No [three]*)
0, 8:
    begin(*6*)(*IND(R), INDR(R), INI(B), INIR(R)*)
    retvalrea(1, seven, val);(*address to transfer the *)
        (* input data *)
    common:
    for j := 1 to times do begin(*7*)
        message(1 div 2, tempo);(*returns the value *)
            (* from the screen *)
        setmem(val, 1, 1, tempo);
        val := val + length
        (*decrement or increment the address of the memory*)
    end; (* 7 *)
    setreq(1, val, seven)
    end;(*set the register to the new address*) (*6*)
2, 10:
    begin(*8*)(*OUID(B), OUTI(B), UDR(R), UIR(B)*)
    retvalrea(1, three, val);(*address to transfer data *)
        (* to the screen *)
    common:
    if l = 4 then
        bytes := 1;(*two bytes to transfer*)
        for i := 1 to times do begin(*9*)
            for repet := 0 to bytes do begin(*10*)
                map(val + repet, 1, 2, tempo);
                write (chr(trunc(tempo) mod 128))
            end;(*10*)
            val := val + length
            (*decrement or increment address*)
        end; (* 9 *)
        setreq(1, val, three)
        (* set the register to the new value *)

```

```
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
```

```

end (* 8 *)
end (* case four *)
end; (* 4 *)
end; (* case opcode *)
end; (* 1 INPUT *)
```

964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000  
1001  
1002  
1003  
1004  
1005  
1006  
1007

PROCEDURE BITMAN.

The procedure executes the following instructions :

operation	1st ans 2nd half byte	3rd half byte	dst	src
( z flag ← not dst (src) )				
BITB	166	-	R	IM
BIT	167	-	R	IM
BITB	38	<> 0	IR	IM
BIT	39	<> 0	IR	IM
BITB	102	= 0	DA	IM
BIT	103	= 0	DA	IM
BIT	102	<> 0	X	IM
BIT	103	<> 0	X	IM
BITB	38	= 0	R	R
BIT	39	= 0	R	R
( dst (src) ← 0 )				
RESB	162	-	R	IM
RES	163	-	R	IM
RESB	34	<> 0	IP	IM
RES	35	<> 0	IP	IM
RESB	98	= 0	DA	IM
RES	99	= 0	DA	IM
RESB	98	<> 0	X	IM
RES	99	<> 0	X	IM
RESB	34	= 0	R	R
RES	35	= 0	R	R
( dst (src) ← 1 )				

```

1021
1022
1023
1024
1025
1026 SFIR      164      R      IM
1027 SET       165      R      IM
1028 SFIR      36      IR      IM
1029 SFI       37      IR      IM
1030 SFIR      100     DA      IM
1031 SET       101     DA      IM
1032 SEIR      100     X      IM
1033 SFI       101     X      IM
1034 SEIR      36      R      R
1035 SFI       37      = 0     R

```

# PARAMETERS

----- : None.

# GLOBAL VARIABLES

- RIT restricts the range of the operation of the procedure retvalreq to the last 4 LSB of the register.Range 1 or f.
- opcode is the decimal value of the opcode under execution.
- ic is the decimal counter.Range 0..maxmem.

# LOCAL VARIABLES

- dst is the destination operand of the opcode.Range 0..15.
- src is the source operand of the opcode.Range 0..15.
- length denotes the size of the used register.Range 0 or 1.
- l denotes the number of the half bytes in the case of reference to memory.Range 2 or 4.
- low is the LSB of the used register.Range 0 or 8.



```

1084 - value, val are used to retrieve values from
1085 the memory and also to set the memory to the
1086 update value after the operation.
1087 - 'a' denotes that a memory copy has been performed
1088 to the register temporary. Range T or F.
1089
1090

```

```

1091 The procedure performs the following in order:

```

- ```

1092 - decodes the dst and src.
1093 - calls the setlength1 to set the variables lenght and l.
1094 - selects one of the base cases in the case statement :
1095   - BII(B) , RES(R) , SET (R)   R, IM
1096   - BII(B) , RES(R) , SET (R)   DA or X
1097   - BII(B) , RES(R) , SET (R)   IR ,IM or R,R
1098 - in the case of the BII operation performs that by itself.
1099 - in the case of the RES or SET operation calls the procedure
1100   setreset to execute the operation depending on the value
1101   of the parameter seta T or F.
1102 - if it has to set or reset a bit from the memory then copies
1103   the value of the memory to the register temporary , performs
1104   the operation on the register and then sets again the memory
1105   to the update value.
1106

```

```

1107 The procedure is called by : FXFULIF.

```

```

1108 The procedure calls the : procedures setsrdsr, setlength1, upperlow
1109   setindex, resetset, DAorX, map, setreg,
1110   retvalreg.

```

```

1113 *****

```

```

1117 procedure BII'MAN;
1118 var
1119   low, l, length, src, dst: integer;

```

```

1120
1121
1122
1123
1124
1125
1126
1127

```



1141  
1142  
1143  
1144  
1145  
1146  
1147  
1148  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1160  
1161  
1162  
1163  
1164  
1165  
1166  
1167  
1168  
1169  
1170  
1171  
1172  
1173  
1174  
1175  
1176  
1177  
1178  
1179  
1180  
1181  
1182

value, val: real;  
a : boolean;

```

1204 ( *****
1205
1206
1207 PROCEDURE resetset.
1208
1209 FUNCTION
1210 ----- the procedure performs the set or reset operation either
1211 to the destination register or the temporary register
1212 R17 which has a copy of the memory in the case of set
1213 or reset of a bit of the memory.
1214 The main procedure sets the value of dst either to the
1215 destination register or the R17.
1216
1217 PARAMETERS
1218 ----- : by reference - seta denotes if the operation
1219 is set or reset.Range I or F.
1220
1221 GLOBAL VARIABLES
1222 ----- ( LOCAL TO BITMAN )
1223 - low is the LSB of the register Range 0 or b.
1224 - src is the value of the src operand.
1225 - length denotes the length of the used register.
1226 Range 0 or 1.
1227 - value is used to retrieve the value of the R17
1228 after the operation and the to set the memory
1229 - val is the address of the memory.Range 0..maxmem.
1230 - a denotes that R17 has been set with a memory
1231 value .Range I or F.
1232
1233 LOCAL VARIABLES
1234 ----- : None.
1235
1236 The procedure is called by : BITMAN.
1237 The procedure calls : the procedures retvalreq,setmem.
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247

```

```

1261
1262
1263
1264
1265 *****
1266 *****
1267 *****
1268 *****
1269 *****
1270 *****
1271 *****
1272 *****
1273 *****
1274 *****
1275 *****
1276 *****
1277 *****
1278 *****
1279 *****
1280 *****
1281 *****
1282 *****
1283 *****
1284 *****
1285 *****
1286 *****
1287 *****
1288 *****
1289 *****
1290 *****
1291 *****
1292 *****
1293 *****
1294 *****
1295 *****
1296 *****
1297 *****
1298 *****
1299 *****
1300 *****
1301 *****
1302 *****

procedure resetset(seta: boolean);

begin (* 1 *)
  if seta then
    rldstlflow + src1 := true (* sets the bit *)
  else
    rldstlflow + src1 := false; (* resets the bit *)
  if a then begin(*2*) (*update the memory*)
    retvalreg(length, temporary, value); (* value ← [R17] *)
    setmem(val, 1, 1, value) (* mem[val] ← value *)
  end(*2*)
end;(* 1 resetset *)

begin(*1*)
  setsrddst(dst, src); (* sets the dst and src *)
  setlength1(length, 1); (* sets the variables length and 1 *)
  a := false;
  case opcode of
    162, 163, 164, 165, 166, 167:
      begin(*RIT(B) RES(B) SET(B) R,IM*)(*2*)
        upperlow(dst, length, 1, low);(*low ← LSR OF THE REGISTER*)
        setindex(length, dst);(* sets dst to the correct value *)
        ic := ic + 1;
        case opcode of
          166, 167:
            rldstlflow := not rldstlflow + src1;(*z← not dst(src)*)
          162, 163:
            resetset(false); (* RES(B) R,IM *)
          164, 165:
            resetset(true) (* SET(B) R,IM *)
        end (* case *)
      end; (*2*)
  end;
end;

```

```

1324 98, 99, 100, 101, 102, 103:
1325 begin(*RIT(B),RES(B), SET(B) DA or X*)(*3*)
1326   DAorX(dst, val); (* val ← address *)
1327   map(val, 1, 1, value); (* value ← mem[val] *)
1328   setreg(1, value, temporary);(*R17← value*)
1329   dst := temporary;
1330   a := true; (* reference to memory *)
1331   case opcode of
1332     102, 103:
1333       r[fcw]lz := not r[temporary][src];
1334     98, 99:
1335       resetset(false); (* RES(B) DA or X *)
1336     100, 101:
1337       resetset(true) (* SET(b) DA or X *)
1338       end (* case *)
1339   end; (*3*)
1340 34, 35, 36, 37, 38, 39:
1341 begin(*RIT(B),RES(B),SET(R) IR, IM or R,R*)(*4*)
1342   if dst <> 0 then begin(*5*) (* IR, IM *)
1343     retvalreg(1, dst, val); (* val ← ldstj *)
1344     map(val, 1, 1, value); (* value ← mem[val] *)
1345     setreg(length, value, temporary);(*R17 ← value*)
1346     dst := temporary;
1347   a := true; (* reference to the memory *)
1348   ic := ic + 1
1349   end else begin(*5,6*) (* R, R *)
1350     map(ic + 1, 1, 1, val);(* val ← number of the register*)
1351     dst := trunc(val);(*number of register*)
1352     RIT := true;(*R,R*)
1353     retvalreg(length, src, value);(*value←Ksrc 0..4 bits*)
1354     RII := false;
1355     src := trunc(value);(*number of bit for test*)
1356     underlow(dst, length, 1, low);(* low ← LSR of the reg.*)
1357     setindex(length, dst);(* dst ← correct number of the *)
1358                                     (* register *)
1359 1360
1361
1362
1363
1364
1365
1366
1367

```

```

1381      ic := ic + 3
1382      end;(*b*)
1383      case opcode of
1384      38, 39:
1385          r[fcw]lz := not r[dst]flow + src);
1386      34, 35:
1387          resetset(false); (* RFS(B) IR,IM or R,R *)
1388      36, 37:
1389          resetset(true) (* SET(B) IR,IM or R,R *)
1390      end (* case *)
1391      end (* 4 *)
1392      end (* case *)
1393      end;(* 1 BITMAN *)

```

1444  
1445  
1446  
1447  
1448  
1449  
1450  
1451  
1452  
1453  
1454  
1455  
1456  
1457  
1458  
1459  
1460  
1461  
1462  
1463  
1464  
1465  
1466  
1467  
1468  
1469  
1470  
1471  
1472  
1473  
1474  
1475  
1476  
1477  
1478  
1479  
1480  
1481  
1482  
1483  
1484  
1485  
1486  
1487

PROCEDURE CALLREI.

The procedure executes the following instructions :

FUNCTION

| operation | 1st and 2nd<br>half bytes | 3rd<br>half<br>bytes | 4th<br>half<br>bytes |
|-----------|---------------------------|----------------------|----------------------|
| CALL IR   | 31                        | -                    |                      |
| CALL DA   | 95                        | = 0                  |                      |
| CALL X    | 95                        | <> 0                 |                      |
| JP IR     | 30                        | -                    | CC                   |
| JP DA     | 94                        | = 0                  | CC                   |
| JP X      | 94                        | <> 0                 | CC                   |
| RFI       | 158                       | = 0                  | CC                   |
|           | 1st half<br>byte          | 2-3-4 th half bytes  |                      |
| CALR RA   | 208<br>(13*16)            | DISPLACEMENT         |                      |
|           | 2nd half<br>byte          | 3-4 half<br>byte     |                      |
| DJNZ      | 240<br>(15*16)            | r                    | DISPLACEMENT         |
| JR RA     | 224<br>(14*16)            | CC                   | DISPLACEMENT         |

The procedure performs the following in order :

- decodes the dst and src operands.
- selects a case of the case statement as follows :
- JP CC, CALL IR, or DA or X

```

1501
1502
1503
1504
1505 -----
1506 - in the case of CALL IR or JP IR retrieves the contents
1507 of the IR register to find the address else calculates
1508 the DA or X address calling the procedure DAorX.
1509 - in the case of CALL IR or CALL DA or X calls the pro-
1510 cedure pushstack to perform the operation of calling.
1511 - in the case of JP IR it checks if the condition is
1512 satisfied and alters the IC to the new address if it
1513 is satisfied else increases the IC to the next instru-
1514 ction.
1515 - CALR RA , DJN7 , JR RA
1516 -----
1517 - decodes the values of the 2nd and 3-4 half bytes.
1518 - selects one of two subcases
1519   - JR or DJNZ
1520 -----
1521   - if the value of the 3-4 half byte is GT 64 then
1522     the displacement is negative or it is a word
1523     register in the case of DJNZ.
1524   - if the opcode is DJNZ (240) retrieves the contents of
1525     the register ,decrements his value and sets the register
1526     to the new value.
1527   - checks if the value of the register equal to zero and
1528     if it equals sets the flag 'a'.
1529   - jumps to the new address setting the IC if the flag 'a'
1530     is true.
1531   - if the opcode is JR checks if the test condition is sa-
1532     tisfied and if it is satisfied then calculates the new
1533     address depending in the sign of the displacement.
1534   - CALR
1535 -----
1536 - calculates the displacement .
1537 - checks if it is negative.
1538 - calculates the new address.
1539 - calls the procedure pushstack to perform the operation.
1540
1541
1542

```



```

1564 - RET
1565 -----
1566 - checks if the test condition is satisfied .
1567 - retrieves the contents of the current nspoff register.
1568 - increases the stackpointer.
1569 - sets the nspoff to his new value and sets the IC.
1570 - checks for overflow of the stack.
1571 - the last four operations are executed only if the test
1572 condition is satisfied.
1573
1574
1575 PARAMETERS
1576 ----- : None.
1577
1578 GLOBAL VARIABLES
1579 ----- :
1580 - opcode is the current value of the opcode
1581 under execution.
1582 - ic is the decimal counter.Range 0..maxmem.
1583
1584 LOCAL VARIABLES
1585 ----- :
1586 - x is used to retrieve to the contents of the
1587 nspoff register.Range 0..maxmem.
1588 - val is used
1589 - as the new address in the case of the opcodes
1590 31,05,30,94.
1591 - as the value of the second half byte in the
1592 case of the opcodes 208,240,274 and also to
1593 calculate the offset address in the case of
1594 the opcode 20A.
1595 - value is used to retrieve the value of the 3-4
1596 th half bytes of the opcode.Range 0..256.
1597 - temp is the contents of the register in the
1598 case of he opcode 0JNZ.Range 0..maxinteg.
1599 - nsp is the value of the stackpointer ,it is
1600 copy of the variable 'x'.Range 0..maxmem.
1601 - length denotes if the used register, in the
1602
1603
1604
1605
1606
1607

```

```

1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662

```

case of the instruction DJNZ, is a word or  
 byte.Range 0..1.  
 - src is the destination operand .Range 0..15.  
 It is a copy of the variable val.  
 - dst is the destination operand .Range 0..15.  
 - a denotes that the cc condition for jump  
 or return is satisfied.Range I or F.  
 - b denotes that the displacement is negative.  
 Range I or F.

The procedure is called by : EXECUIF.  
 The procedure calls : the procedures retvalreg,DAorX, testcc  
 map,pushstack,setreg,error.

\*\*\*\*\*)

```

procedure CALLKEL;
var
  x, val, value, temp: real;
  nsp, length, src, dst: integer;
  a, h: boolean;

```

```

1684 *****
1685 ( *****
1686 *****
1687 *****
1688 *****
1689 *****
1690 *****
1691 *****
1692 *****
1693 *****
1694 *****
1695 *****
1696 *****
1697 *****
1698 *****
1699 *****
1700 *****
1701 *****
1702 *****
1703 *****
1704 *****
1705 *****
1706 *****
1707 *****
1708 *****
1709 *****
1710 *****
1711 *****
1712 *****
1713 *****
1714 *****
1715 *****
1716 *****
1717 *****
1718 *****
1719 *****
1720 *****
1721 *****
1722 *****
1723 *****
1724 *****
1725 *****
1726 *****
1727 *****
1728 *****
1729 *****
1730 *****
1731 *****
1732 *****
1733 *****
1734 *****
1735 *****
1736 *****
1737 *****
1738 *****
1739 *****
1740 *****
1741 *****
1742 *****
1743 *****
1744 *****
1745 *****
1746 *****
1747 *****
1748 *****
1749 *****
1750 *****
1751 *****
1752 *****
1753 *****
1754 *****
1755 *****
1756 *****
1757 *****
1758 *****
1759 *****
1760 *****
1761 *****
1762 *****
1763 *****
1764 *****
1765 *****
1766 *****
1767 *****
1768 *****
1769 *****
1770 *****
1771 *****
1772 *****
1773 *****
1774 *****
1775 *****
1776 *****
1777 *****
1778 *****
1779 *****
1780 *****
1781 *****
1782 *****
1783 *****
1784 *****
1785 *****
1786 *****
1787 *****
1788 *****
1789 *****
1790 *****
1791 *****
1792 *****
1793 *****
1794 *****
1795 *****
1796 *****
1797 *****
1798 *****
1799 *****
1800 *****
1801 *****
1802 *****
1803 *****
1804 *****
1805 *****
1806 *****
1807 *****
1808 *****
1809 *****
1810 *****
1811 *****
1812 *****
1813 *****
1814 *****
1815 *****
1816 *****
1817 *****
1818 *****
1819 *****
1820 *****
1821 *****
1822 *****
1823 *****
1824 *****
1825 *****
1826 *****
1827 *****
1828 *****
1829 *****
1830 *****
1831 *****
1832 *****
1833 *****
1834 *****
1835 *****
1836 *****
1837 *****
1838 *****
1839 *****
1840 *****
1841 *****
1842 *****
1843 *****
1844 *****
1845 *****
1846 *****
1847 *****
1848 *****
1849 *****
1850 *****
1851 *****
1852 *****
1853 *****
1854 *****
1855 *****
1856 *****
1857 *****
1858 *****
1859 *****
1860 *****
1861 *****
1862 *****
1863 *****
1864 *****
1865 *****
1866 *****
1867 *****
1868 *****
1869 *****
1870 *****
1871 *****
1872 *****
1873 *****
1874 *****
1875 *****
1876 *****
1877 *****
1878 *****
1879 *****
1880 *****
1881 *****
1882 *****
1883 *****
1884 *****
1885 *****
1886 *****
1887 *****
1888 *****
1889 *****
1890 *****
1891 *****
1892 *****
1893 *****
1894 *****
1895 *****
1896 *****
1897 *****
1898 *****
1899 *****
1900 *****
1901 *****
1902 *****
1903 *****
1904 *****
1905 *****
1906 *****
1907 *****
1908 *****
1909 *****
1910 *****
1911 *****
1912 *****
1913 *****
1914 *****
1915 *****
1916 *****
1917 *****
1918 *****
1919 *****
1920 *****
1921 *****
1922 *****
1923 *****
1924 *****
1925 *****
1926 *****
1927 *****
1928 *****
1929 *****
1930 *****
1931 *****
1932 *****
1933 *****
1934 *****
1935 *****
1936 *****
1937 *****
1938 *****
1939 *****
1940 *****
1941 *****
1942 *****
1943 *****
1944 *****
1945 *****
1946 *****
1947 *****
1948 *****
1949 *****
1950 *****
1951 *****
1952 *****
1953 *****
1954 *****
1955 *****
1956 *****
1957 *****
1958 *****
1959 *****
1960 *****
1961 *****
1962 *****
1963 *****
1964 *****
1965 *****
1966 *****
1967 *****
1968 *****
1969 *****
1970 *****
1971 *****
1972 *****
1973 *****
1974 *****
1975 *****
1976 *****
1977 *****
1978 *****
1979 *****
1980 *****
1981 *****
1982 *****
1983 *****
1984 *****
1985 *****
1986 *****
1987 *****
1988 *****
1989 *****
1990 *****
1991 *****
1992 *****
1993 *****
1994 *****
1995 *****
1996 *****
1997 *****
1998 *****
1999 *****
2000 *****

```

PROCEDURE pushstack.

FUNCTION

----- The procedure

- retrieves the value of the nspoff register (stackpointer)
- decreases the stackpointer.
- checks for underflow.
- saves the IC (address of return )
- sets the nspoff register to his new value.

PARAMETERS

----- : None.

GLOBAL VARIABLES

-----

(LOCAL IN THE PROCEDURE CALLRET )

- x is used to retrieve the value of the nspoff register.
- nsp is a copy of the x variable in integer value. Range 0..maxmem.
- val is the next address for execution and it is assigned to the variable IC.Range 0..maxmem. (GLOBAL IN MAIN PROGRAM )
- ic the decimal value of the r counter register. The original value is saved to the stack and then copies the value of the variable val. Range 0..maxmem.
- alldone denotes that the user's program is free of error conditions as stack underflow.

the procedure is called by : CALLRET.

the procedure calls : retvalreq,error, setmem , setreq.

```

1741
1742
1743
1744
1745 *****
1746 *****
1747 *****
1748 *****
1749 *****
1750 *****
1751 *****
1752 *****
1753 *****
1754 *****
1755 *****
1756 *****
1757 *****
1758 *****
1759 *****
1760 *****
1761 *****
1762 *****
1763 *****
1764 *****
1765 *****
1766 *****
1767 *****
1768 *****
1769 *****
1770 *****
1771 *****
1772 *****
1773 *****
1774 *****
1775 *****
1776 *****
1777 *****
1778 *****
1779 *****
1780 *****
1781 *****
1782 *****

procedure pushstack;

begin(*1*)
  retvalreg(1, nspoff, x);(*x = K(nspoff)**)
  nsp := trunc(x) - 2;(*decrements the stack pointer*)
  if nsp <= 0 then
    error(2);(*underflow*)
  if not alldone then begin(*2*)
    setmem(nsp, 1, 4, ic);(*saves the address of return*)
    setreq(1, nsp, nspoff);(*sets K(nspoff) to the new value*)
    ic := trunc(val); (* sets the ic to the call address *)
    setreq(1, ic, counter);(* sets the counter register to the *)
                          (* new address *)
    end;(*2*)
  end;(* 1 pushstack *)

begin(*1*)
  setsrcdst(dst, src);
  a := false;
  b := false;
  case opcode of
    31, 95, 30, 94:
      begin(*2*)(*JP CC, CALL IR,DA,x*)
        if (opcode = 31) or (opcode = 30) then begin(*3*)
          retvalreg(1, dst, val);(*val = [Rdst]**)
          ic := ic + 1
          end else(*3*)
            DAorX(dst, val);
          if (opcode = 31) or (opcode = 95) then

```

```

1804 pushstack
1805 else begin(*5*)
1806   testcc(a, src);
1807   if a then
1808     ic := trunc(val)
1809   end;(*5*)
1810 end;(*2*)
1811
1812 208, 240, 224:
1813 begin(*CALLR, DJNZ ,JR*)(*6*)
1814   map(ic, 1, 2, value);(*3,4 bytes*)
1815   map(ic - 1, 1, 1, val);(*2 nd byte*)
1816   ic := ic + 1;
1817   case opcode of
1818     240, 224:
1819       begin(*7*)(*JR,DJNZ*)
1820         if value >= 64 then begin(*8*)
1821           b := true;(*negative*)
1822           length := 1 (* word register *)
1823         end else (*8*)
1824           length := 0; (* byte register *)
1825         if opcode = 240 then begin(*9*) (* DJNZ *)
1826           src := trunc(val);(*register*)
1827           if b then
1828             (* subtracts 64 in order to delete the bit that *)
1829             (* denotes that the used register is word type *)
1830             value := value - 64;
1831             retvalreg(length, src, temp);(* temp ← [src] *)
1832             temp := temp - 1;
1833             if temp <= 0 then begin(*10*)
1834               temp := 0;
1835               a := true (* sets the a flag *)
1836             end;(*10*)
1837             setreq(length, temp, src);(*Rsrc←temp*)
1838             if not a then (* no zero so jump *)
1839               ic := ic - trunc(2 * value)
1840
1841
1842
1843
1844
1845
1846
1847

```

```

1861
1862
1863
1864
1865     end else begin(*9,10*)(*JR cc*)
1866         testcc(a, val); (* tests the test condition *)
1867         if a then begin(*11*)
1868             if b then(*negative*)
1869                 value := value - 256;
1870             ic := ic + trunc(2 * value)
1871             end(*11*)
1872         end(*10*)
1873     end;(*7*)
1874
1875 208:     begin(*CALR*)(*11*)
1876         val := val * 256 + value; (* displacement are the *)
1877             (* 2-3-4 half bytes *)
1878         if val > 2047 then (* negative value *)
1879             val := val - 4096;
1880             val := val * 2 + ic;
1881             pushstack
1882             end (* 11 *)
1883         end(*case*)
1884     end; (* 6 *)
1885
1886 158:     begin(*RET*)(*13*)
1887         ic := ic + 1;
1888         testcc(a, src);
1889         if a then begin(*14*)
1890             retvalreg(1, nspoff, x);(*x - Rfnsppoff1*)
1891             nsp := trunc(x) + 2;(*increments stackpointer*)
1892             setren(1, nsp, nspoff);(*sets nspoff register to the *)
1893                 (* new value *)
1894             if nsp - 2 <= maxmem then begin(*15*)
1895                 map(nsp - 2, 1, 4, val);
1896                 ic := trunc(val);(*restores the address*)
1897                 setren(1, ic, counter)
1898             end else(*sets counter register to the new value*)(*15*)
1899                 error(2)(*overflow of the stack*)
1900
1901

```

1925           end(\*14\*)  
1926        end (\* 13 \*)  
1927        end (\* case \*)  
1928        end;(\* 1 CALLRET \*)  
1929  
1930  
1931  
1932  
1933  
1934  
1935  
1936  
1937  
1938  
1939  
1940  
1941  
1942  
1943  
1944  
1945  
1946  
1947  
1948  
1949  
1950  
1951  
1952  
1953  
1954  
1955  
1956  
1957  
1958  
1959  
1960  
1961  
1962  
1963  
1964  
1965  
1966  
1967



```
1981
1982
1983
1984
1985 (*****
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
```

```

PROCEDURE DIVMULT.

FUNCTION
----- The procedure executes the following instructions :
operation      1-2 nd      3rd
                half bytes  half byte
-----
1996 DIV R,R      155      -
1997 DIVL R,R     154      -
1998 DIV R,IM     27      = 0
1999 DIVL R,IM    26      = 0
2000 DIV R,IM     91      = 0
2001 DIVL R,IM    90      = 0
2002 DIV R,X      91      <> 0
2003 DIVL R,X     90      <> 0
2004 MULT R,R     153      -
2005 MULTL R,R    152      -
2006 MULT R,IM    25      = 0
2007 MULTL R,IM   24      = 0
2008 MULT R,IR    19      <> 0
2009 MULTL R,IR   18      <> 0
2010 MULT R,DA    59      = 0
2011 MULTL R,DA   58      = 0
2012 MULT R,X     59      <> 0
2013 MULTL R,X    58      <> 0
```

The procedure performs the following :

- sets the variables length and l.
- checks if the multiplicand or the dividend is negative. If it is then the procedure positive changes the operand to positive , setting the flag a to true and it retrieves his

```

2044 value.
2045 - checks if the multiplier or the divisor is negative. If it
2046 is then the procedure positive changes that to positive
2047 , retrieves his positive value and sets the flag b to true.
2048 If the operand is in the memory then the procedure uses the
2049 the register R17 for the above operations. If the operand is
2050 a register after the operation changes the register to his
2051 original value.
2052 - for the DIV it checks if the divisor is <> 0 and less from
2053 the dividend.
2054 FOR THE CASE OF DIV
2055 -----
2056 - performs the operation
2057 - checks if the quotient is less than the maxinterg.
2058 - calculates the remainder.
2059 - sets the high order half of the destination register to
2060 the value of the remainder.
2061 - sets the low half order of the destination register to the
2062 value of the quotient.
2063 - if the divisor is negative changes the value of the remainder
2064 to negative.
2065 - if the divisor and dividend have different signs changes
2066 the sign of the quotient.
2067 - sets the PV flag if the divisor equals to zero or the result
2068 does not fit to the low half order of the register .
2069 IN THE CASE OF MUL
2070 -----
2071 - calculates the result.
2072 - in the case when the result is greater than the maxinteg
2073 then in order to avoid overflow in the integers of the UNIX
2074 system sets first the high half part of the register and then
2075 the low half part of the register.
2076 - if the two operands have different sign changes the sign of
2077 the result.
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087

```

```

2101
2102
2103
2104
2105 PARAMETERS
2106 ----- : None.
2107
2108 GLOBAL VARIABLES
2109 ----- : - opcode is the current value of the opcode under
2110           execution.
2111           - ic is the decimal counter.Range 0..maxmem.
2112
2113 LOCAL VARIABLES
2114 ----- : - x1 is the remainder of the division.
2115           - val is used to retrieve values from the memory
2116           and register and final has the value of the divi-
2117           sor or multiplier.
2118           - value has the value of the dividend or multiplicand.
2119           - l denotes the number of half bytes in the case
2120           of reference to the memory.Range 2 or 4 or 8.
2121           - length denotes the size of the used register.
2122           Range 1 or 2.
2123           - 'a' denotes that the multiplicand or the dividend
2124           are negative.Range T or F.
2125           - 'b' denotes that the multiplier or the divisor
2126           are negative .Range T or F.
2127
2128 the procedure is called by : FxFCUIF.
2129 the procedure calls : the procedures setlength1, positive, onetwocompl
2130                     IMorIR, setreg, map.
2131
2132
2133 *****
2134
2135
2136 procedure DIVMULT;
2137 var
2138   x1, val, value, temp: real;
2139   l, length: integer;
2140
2141 *****

```

a, b: boolean;

2164  
2165  
2166  
2167  
2168  
2169  
2170  
2171  
2172  
2173  
2174  
2175  
2176  
2177  
2178  
2179  
2180  
2181  
2182  
2183  
2184  
2185  
2186  
2187  
2188  
2189  
2190  
2191  
2192  
2193  
2194  
2195  
2196  
2197  
2198  
2199  
2200  
2201  
2202  
2203  
2204  
2205  
2206  
2207

```

2221
2222
2223
2224
2225  (*****
2226
2227      PROCEDURE positive.
2228
2229  FUNCTION
2230  ----- The procedure checks if the passed as parameter register
2231           is negative and if it is changes his value to positive
2232           and retrieves his value
2233
2234  PARAMETERS
2235  ----- : by value      - index is the number of the register.
2236           Range 0..15.
2237           - length is the size of the register.
2238           Range 1 or 2.
2239           by reference  - a denotes that the value of the register
2240                       - is negative.Range T or F.
2241                       - value is the absolute value of the re-
2242                         gister.Range 0..maxinteq.
2243
2244  GLOBAL VARIABLES
2245  ----- : None.
2246
2247  LOCAL VARIABLES
2248  ----- : The parameters.
2249
2250  The procedure is called by : DIVMULT.
2251  The procedure calls : the procedures onetwocompl,retvalreq.
2252
2253  *****
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
3000

```

```

2284 if rindex[15] then begin(*2*)(*negative*)
2285 onetwocompl(index, length, 2, false);
2286 a := true
2287 end;(*2*)
2288 retvalreg(length, index, value)(*value + [kdst]*)
2289 end;(*1 positive*)
2290
2291
2292
2293 begin(*1*)
2294 a := false;
2295 b := false;
2296 x1 := 0;
2297 temp := 0;
2298 setlength1(length, 1);
2299 case opcode of
2300 153, 152, 25, 24, 88, 89: (*MULT (L)*)
2301   positive(dst + length, length, a, value);
2302 155, 154, 27, 26, 91, 90: (*DIV (L) *)
2303   positive(dst, 2 * length, a, value);
2304 end;(*case*)
2305 case opcode of
2306 155, 154, 153, 152:
2307   begin(*DIV(L), MULT(L) R,R*)(*2*)
2308     positive(src, length, b, val);
2309     if b then
2310       onetwocompl(src, length, 2, false);(*return to original *)
2311       (* value the register *)
2312     ic := ic + 1
2313   end;(*2*)
2314 27, 26, 25, 24:
2315   begin(*DIV(L), MULT(L) IM or IR*)(*3*)
2316     IMorIR(src, 1, length, val); (* val + address *)
2317     setreg(length, val, temporary);(*R17 + val*)
2318     positive(temporary, length, b, val)
2319     (* val is the final absolute value of the multiplier *)
2320
2321
2322
2323
2324
2325
2326
2327

```



```

2341      (* or the divisor *)
2342      end;(*3*)
2343      91, 90, 89, 88:
2344      begin(*DIV(L),MULT(L) (DA or X)*) (*b*)
2345      DAorX(src, val); (* val is the address *)
2346      map(val, 1, 1, val);(*val ← mem[val]*
2347      setreq(length, val, temporary);(*R17 ← val*)
2348      positive(temporary, length, b, val)
2349      (* val is the final absolute value of the multiplier or *)
2350      (* divisor *)
2351      end (* 6 *)
2352      end; (*case*)
2353      case opcode of
2354      155, 154, 27, 26, 91, 90:
2355      begin(*7*)(*DIV(L)*)
2356      if (val <> 0) and (val <= value) then begin(*8*)
2357      temp := value / val; (* makes the division *)
2358      if temp <= maxinteg then begin(*9*)
2359      setreq(length, trunc(temp), dst + length);(*Low R ← *)
2360      (* quotient *)
2361      x1 := (temp - trunc(temp)) * val; (* calculates the *)
2362      (* remainder *)
2363      if abs(x1 - trunc(x1)) > 0.5 then
2364      x1 := x1 + 1;
2365      setreq(length, trunc(x1), dst);(*upper R ← remainder*)
2366      if a and not b or not a and b then begin(*10*)
2367      (* the two operands have different signs so the result
2368      must be negative *)
2369      onetwocompl(dst + length, length, 2, false);(*negative *)
2370      rffcwlsl := true (* sets the sign flag *)
2371      end else(*10*)
2372      rffcwlsl := false;
2373      if b then
2374      (* the divisor is negative so the remainder is negative*)
2375      onetwocompl(dst, length, 2, false)(*negative remainder*)
2376
2377
2378
2379
2380
2381
2382
2383

```



```

2404 end (* 9 *)
2405 end(*8*);
2406 r[fcw]l[pl] := (val = 0) or (temp > maxinteg)
2407 and (length = 2) or (temp >= 32768) and (length = 1)
2408
2409 end;(*7*)
2410 155, 152, 25, 24, 89, 88:
2411 begin(*11*) (*MULT(L)**)
2412   temp := value * val;
2413   if temp/2 > maxinteg then begin(*12*)
2414     setreq(length,temp -2*(maxinteg+1)* trunc(temp/(2*(maxinteg+1))),
2415       dst + length);
2416     (* sets the upper half of the register *)
2417     setreq(length, trunc(temp/(2*(maxinteg+1))), dst)
2418     (* sets the low half of the register *)
2419     end else(*12*)
2420     setreq(2 * length, temp, dst);
2421     if a and not b or not a and b then begin(*13*)
2422       (* the two operands have different signs so the result
2423         must be negative *)
2424       onetwocompl(dst, 2 * length, 2, false);
2425       r[fcw]l[s] := true
2426     end else(*13*)
2427       r[fcw]l[s] := false;
2428       r[fcw]l[pl] := false
2429     end (* 11 *)
2430   end; (*case*)
2431   r[fcw]l[c] := (length = 1) and (temp >= 32768)
2432     or (length = 2) and (temp >= maxinteg +1);
2433   r[fcw]l[z] := (temp = 0) or (val = 0)
2434
2435 end;(* 1 DIVMULT *)

```



|      |      |       |     |      |    |
|------|------|-------|-----|------|----|
| 2524 | INCB | R,IM  | 168 | R    | IM |
| 2525 | INC  | R,IM  | 169 | R    | IM |
| 2526 | INCB | IR,IM | 40  | <> 0 | IM |
| 2527 | INC  | IR,IM | 41  | <> 0 | IM |
| 2528 | INCB | DA,IM | 104 | = 0  | IM |
| 2529 | INC  | DA,IM | 105 | = 0  | IM |
| 2530 | INC  | DA,IM | 105 | = 0  | IM |
| 2531 | INC  | DA,IM | 105 | = 0  | IM |
| 2532 | INCB | X,IM  | 104 | <> 0 | IM |
| 2533 | INC  | X,IM  | 105 | <> 0 | IM |
| 2534 | SURB | R,R   | 130 | K    | R  |
| 2535 | SUR  | R,R   | 131 | R    | R  |
| 2536 | SURL | R,R   | 140 | R    | R  |
| 2537 | SURB | IM,R  | 02  | = 0  | R  |
| 2538 | SUR  | IM,R  | 03  | = 0  | R  |
| 2539 | SURL | IM,R  | 10  | = 0  | R  |
| 2540 | SURB | IR,R  | 02  | = 0  | R  |
| 2541 | SUR  | IR,R  | 03  | <> 0 | R  |
| 2542 | SURL | IR,R  | 10  | <> 0 | R  |
| 2543 | SURB | DA,R  | 66  | = 0  | R  |
| 2544 | SUR  | DA,R  | 67  | = 0  | R  |
| 2545 | SURL | DA,R  | 82  | = 0  | R  |
| 2546 | SURB | X,R   | 66  | <> 0 | R  |
| 2547 | SUR  | X,R   | 67  | <> 0 | R  |
| 2548 | SURL | X,R   | 82  | <> 0 | R  |
| 2549 | SBCB | R,R   | 182 | R    | R  |
| 2550 | SBC  | R,R   | 183 | R    | R  |
| 2551 | CPR  | R,R   | 138 | R    | R  |
| 2552 | CP   | R,R   | 139 | K    | R  |
| 2553 | CPL  | R,R   | 144 | K    | R  |
| 2554 | CPR  | IM,R  | 10  | <> 0 | R  |
| 2555 | CP   | IM,R  | 11  | <> 0 | R  |
| 2556 | CPL  | IM,R  | 10  | <> 0 | R  |
| 2557 | CPR  | IR,K  | 10  | <> 0 | R  |
| 2558 | CP   | IR,R  | 11  | <> 0 | R  |
| 2559 | CPL  | IR,R  | 10  | <> 0 | R  |
| 2560 |      |       |     |      |    |
| 2561 |      |       |     |      |    |
| 2562 |      |       |     |      |    |
| 2563 |      |       |     |      |    |
| 2564 |      |       |     |      |    |
| 2565 |      |       |     |      |    |
| 2566 |      |       |     |      |    |
| 2567 |      |       |     |      |    |



2644  
2645  
2646  
2647  
2648  
2649  
2650  
2651  
2652  
2653  
2654  
2655  
2656  
2657  
2658  
2659  
2660  
2661  
2662  
2663  
2664  
2665  
2666  
2667  
2668  
2669  
2670  
2671  
2672  
2673  
2674  
2675  
2676  
2677  
2678  
2679  
2680  
2681  
2682  
2683  
2684  
2685  
2686  
2687

- EXTISB denotes that the operation is EXTISB and extends the low bytes registers feature to all the registers.Range T or F.

# LOCAL VARIABLES

- val,value , temp as described in each subcase.
- l denotes the number of the half bytes in the case of reference to the memory.Range 2,4 or 8.
- length denotes the length of the used register. Range 0..2.
- subst denotes that the performed operation is subtraction and it is used to change the C flag.Range T or F.
- zero denotes that the subtractor is zero and so does not affect the falg.Range T or F.
- 'a' is used to save the value of the C flag in the case of execution of the DEC or INC opcode.

The procedure is called by : EXECUTE.

The procedure calls : the procedures setlength,l,MorlR,setreg,ADD, DAorX, onetwocompl, map, retvalreq, setmem, testcc,setindex,upperlow.

\*\*\*\*\*)

## procedure ARITHMETIC;

var  
val, value, temp: real;  
l,l1, length: integer;

```

2701 arzero,subst : boolean;
2702
2703 begin(x1*)
2704   setlength1(length, 1);
2705   (* sets the variables length, 1 *)
2706   subst := false;
2707   zero := false; (* reset the zero flag *)
2708   a:=r[fcw/4]; (* saves the C flag*)
2709   (* sets to addition the flag *)
2710   if ( opcode = 0 ) and ( src = 0 ) and ( dst = 0 ) then
2711     allone := true;(* no code to execute*)
2712   SUBB:= false;
2713   ADDR := false;
2714   (* resets the flags in each call of the procedure *)
2715   if (opcode =130) or (opcode =2) or (opcode = 60)
2716     or (opcode = 182 ) then
2717     SUBB := true;(*SUBB or SURC operation will be executed*)
2718   if (opcode=128) or (opcode =0) or (opcode =64)
2719     or (opcode =180) then
2720     ADDR := true;(*ADDB or ADDBC operation will be executed*)
2721   case opcode of
2722     0, 1, 2, 3, 8, 18, 22:
2723     begin(*2*)(*ADD (R,L) IM or IR, R,Rdst+Rdst+src*)
2724       (*SUB (R,L) IM or IR,R, Rdst +Rdst-src*)
2725       IMorIP(src, 1, length, val);
2726       (* assigns to the variable val the data *)
2727       zero := val = 0; (* sets the flag if the source
2728         equals to zero *)
2729       setren(length, val, temporary);(*R17 ← val*)
2730       (* sets the R17 with the data *)
2731       if (opcode = 2) or (opcode = 3) or (opcode = 18) then
2732         onetwocompl(temporary, length, 2, false);(*R17 ← -R17*)
2733       (* if the operation is subtraction performs
2734         two complement to the R1/ *)
2735       ADD(dst, temporary, length, true, false);
2736
2737
2738
2739
2740
2741
2742

```



```

2764 (* performs addition or subtraction depending on
2765 the status of the R17 *)
2766
2767 end;(*2*)
2768
2769 64, 65, 66, 67, 82:
2770 begin(*6*)(*ADD(B,L) DA or X,R ,Rdst +Rdst+value*)
2771 (*SUB (B,L) DA or X,R, Rdst + Rdst - value*)
2772 DAorX(src, val);
2773 (* assigns to the variable val the address of the source
2774 operand *)
2775 map(val, 1, 1, value);(*value +mem[val]*)
2776 (* assigns to the variable value the data *)
2777 zero := value = 0; (* sets the flag if the source
2778 equals to zero *)
2779 setreq(length, value, temporary);(*R17 + value*)
2780 (* sets the R17 with the data *)
2781 if (opcode = 66) or (opcode = 67) or (opcode = 82) then
2782 onetwocompl(temporary, length, 2, false);(*R17 + - R17*)
2783 (* if the opcode is subtraction performs two complement
2784 to the R17 *)
2785 ADD(dst, temporary, length, true, false);
2786 (* performs addition or subtraction depending on the
2787 status of the R17 *)
2788
2789 end;(*6*)
2790 180, 181:
2791 begin(*9*)(*ADC (B) Rdst + Rdst + Rdst + carry*)
2792 ADD(dst, src, length, true, true);
2793 (* adds with carry the dst and src operands *)
2794 ic := ic + 1
2795
2796 end;(*9*)
2797 128, 129, 150:
2798 begin(*10*)(*ADD (B,L) Rdst + Rdst + Rsrc*)
2799 ADD(dst, src, length, true, false);
2800 (* adds the src and dst operands without carry *)
2801 ic := ic + 1
2802
2803 end;(*10*)

```



```

2821      168, 169, 170, 171:
2822      begin(*11*)(*INC (b) Rdst ← Rdst + IM*)
2823      (*DEC (B) Rdst ← Rdst - IM*)
2824      setreq(length, dst + 1, temporary);(*src,dst inverse*)
2825      (* sets the R17 with IM data *)
2826      if (opcode = 170) or (opcode = 171) then
2827      onetwocompl(temporary, length, 2, false);(*R17 ← - R17*)
2828      (* if the opcode is decrement then performs two
2829      complement to the R17 *)
2830      ADD(src, temporary, length, true, false);
2831      (* adds or subtracts the IM data from the dst register
2832      depending on the status of the R17 *)
2833      rffcwl[c] := a;
2834      ic := ic + 1
2835      end; (*11*)
2836      40, 41, 42, 43:
2837      begin(*12*)(*INC (B) IR,memory*)
2838      (*DEC (B) IR,memory*)
2839      retvalreq(1, src, value);(*value ← [Rdst]*)
2840      (* assigns the destination address to the variable
2841      value *)
2842      map(value, 1, 1, temp);(*temp←mem[value]*)
2843      (* assigns the destination data to the variable
2844      temp *)
2845      setreq(length, temp, temporary);(*R17 ← temp*)
2846      (* sets the R17 with the destination data *)
2847      setreq(length, dst + 1, temporary + 1);(*R18 ← src*)
2848      (* sets the R18 with the source data *)
2849      if (opcode = 42) or (opcode = 43) then
2850      onetwocompl(temporary + 1, length, 2, false);
2851      (* R18 ← - R18*)
2852      (* if the operation is decrement then performs two
2853      complement to the R18 *)
2854      ADD(temporary, temporary + 1, length, true, false);
2855      (* R17 ← R17 + R18 *)
2856
2857
2858
2859
2860
2861

```

```

2884 (* performs addition or subtraction depending
2885 on the status of the R18 *)
2886 rffcw[c] := a; (* restores the C flag *)
2887 retvalreq(length, temporary, temp); (* temp ← R17 *)
2888 (* assigns to the variable temp the result of the
2889 operation *)
2890 setmem(value, 1, 1, temp);
2891 (* sets the destination memory with the result *)
2892 ic := ic + 1
2893 end; (* 12 *)
2894 104, 105, 106, 107:
2895 begin (* 14 *) (* INC (B) DA or X, MEM *)
2896   (* DEC (B) DA or X, MEM *)
2897   DAorX(src, val);
2898   (* assigns to the variable val the address of the
2899   destination *)
2900   map(val, 1, 1, temp); (* temp ← mem[val] *)
2901   (* assigns to variable temp the destination data *)
2902   setreq(length, temp, temporary); (* R17 ← temp *)
2903   (* sets the R17 with the destination data *)
2904   setreq(length, dst + 1, temporary + 1); (* R18 ← src *)
2905   (* sets the R18 with the source data *)
2906   if (opcode = 106) or (opcode = 107) then
2907     onetwocompl(temporary + 1, length, 2, false);
2908     (* R18 ← - R18 *)
2909   (* if the operation is decrement then performs two
2910   complement to the R18 *)
2911   ADD(temporary, temporary + 1, length, true, false);
2912   (* R17 ← R17 + R18 *)
2913   (* performs addition or subtraction depending on the
2914   status of the R18 *)
2915   rffcw[c] := a;
2916   retvalreq(length, temporary, temp); (* temp ← R17 *)
2917   (* assigns to the variable temp the result of the
2918   operation *)

```

```

2941      setmem(trunc(val), 1, 1, temp)
2942      (* sets the destination memory with the result *)
2943      end;(*14*)
2944      130, 131, 146:
2945      begin(*17*)(*SUB (B,L) R,R*)
2946          subst := true;
2947          retvalreg (length,src,val); (*retrieves the contents
2948          of the source register to check if it is zero *)
2949          zero := val = 0; (* sets the flag if equal to zero *)
2950          onetwocompl(src, length, 2, false);(*Rsrc ← -Rsrc*)
2951          (* performs two complement to the source register*)
2952          AND(dst, src, length, true, false);(*Rdst ← Rdst+Rsrc*)
2953          (* performs addition *)
2954          onetwocompl(src, length, 2, false);(*- Rsrc ← Rsrc*)
2955          (* restores the source register*)
2956          ic := ic + 1
2957      end;(*17*)
2958      182, 183:
2959      begin(*18*)(*SUB (B) Rdst ←Rdst - src-carry*)
2960          subst := true;
2961          retvalreg ( length,src,val); (* retrieves the contents
2962          of the source register to check if it equals to zero *)
2963          zero := val = 0; (* sets the flag*)
2964          onetwocompl(src, length, 2, false);(*Rsrc ← - Rsrc*)
2965          (* performs two complement to the source register *)
2966          AND(dst, src, length, true, true);
2967          (* performs addition with carry *)
2968          onetwocompl(src, length, 2, false);(*- Rsrc ← Rsrc*)
2969          (* restores the source register *)
2970          ic := ic + 1;
2971      end;(*18*)
2972      177:
2973      begin(*EXTS (B,L) dst (R)*)(*19*)
2974          case dst of(*invert dst,src*)
2975              0:
2976
2977
2978
2979
2980
2981

```

```

3004 begin(*20*)(*EXTSB*)
3005 EXTSB := true;
3006 retvalreq(0, src, value);(*value + Rlow byte*)
3007 lol := 0;
3008
3009 setindex(lol,src);
3010 (* assigns the value of the low byte of the
3011 destination register to the variable value *)
3012 EXTSB := false;
3013 if r[src][7] then
3014     value := value + 65280;(*extend sign one *)
3015 (* if the MSB is one then increases the value
3016 of the register in order to be all ones in the
3017 high byte *)
3018 (*else extend the zero*)
3019 setreq(1, value, src)
3020 (* sets the register to his new value *)
3021 end;(*20*)
3022
3023 10:
3024 if r[src][15] then begin(*21*) (*FXTS*)
3025     setreq(1, 1, src-1);
3026 (* checks if the low word register has negative
3027 value *)
3028 (* and if yes then sets the register with the value
3029 one *)
3030 onetwocompl(src-1, 1, 2, false)(*extend the one *)
3031 (* performs two complement to the register and
3032 so the one one is transformed to all ones *)
3033 end else(*21*)
3034     setreq(1,0,src-1);(*extend the zero to*)
3035     (*the upper register*)
3036
3037 7:
3038 if r[src][15] then begin(*22*)
3039     setreq(2, 1, src-2);(*EXTSL*)
3040 (* checks if the low double register has
3041 negative value and if yes then sets the upper
3042
3043
3044
3045
3046
3047

```

```

3061 double register to one *)
3062 onetwocompl(src - 2, 2, 2, false)
3063 (* performs two complement and so the one one
3064 is transformed to all ones *)
3065 end else (* 22 *)
3066 setreg(2,0,src - 2);(*set the upper double*)
3067 (* register to zero *)
3068 end; (* case *)
3069 ic := ic + 1
3070 end;(*19*)
3071 138, 139, 144, 10, 11, 16, 74,
3072 75, 80:
3073 begin(*CP(B,L) dst -src*)(*24*)
3074 subst := true;
3075 retvalreq(length, dst, temp);(*temp ← kdst*)
3076 (* assigns the value of the Rdestination to the
3077 variable temp *)
3078 case opcode of
3079 138, 139, 144:
3080 begin (* 25 *)(*Rdst ← Rsrc*)
3081 retvalreq(length, src, value);(*value ← Rsrc*)
3082 (* assigns the value of the source register to
3083 the variable value*)
3084 ic := ic + 1
3085 end; (* 25 *)
3086 10, 11, 16:
3087 begin(*Rdst ← IM or IR*)(*26*)
3088 IMorIR(src, 1, length, val);
3089 (* assigns the source data to the variable val*)
3090 value := val
3091 (* copies the value of the val to the variable
3092 value *)
3093 end; (* 26 *)
3094 74, 75, 80:
3095 begin(*Rdst ← DA or X*)(*27*)

```



```

3124 DAORX(src, val);
3125 (* assigns the address of the data to the variable
3126    val *)
3127 map(val, 1, 1, value)
3128 (* assigns the value of the data to the variable value*)
3129 end(* 27 *)
3130
3131 end; (* case *)
3132 zero := value = 0; (* sets the flag if the source
3133    data equal to zero *)
3134 setreg(length, value, temporary);(*R17 + value*)
3135 (* sets the R17 with the source data *)
3136 onetwocompl(temporary, length, 2, false);(*R17 + - R17*)
3137 (* performs two complement to the R17 *)
3138 AND(dst, temporary, length, true, false);(*Rdst + Rdst+R17*)
3139 (* subtracts the source register R17 from the destination
3140    register setting the flags*)
3141 setreg(length, temp, dst);(*Rdst + original value*)
3142 (* sets the destination register to his original value*)
3143 end;(*24*)
3144 174, 175:
3145 begin(* ICC(b) cc, dst*)(*32*)
3146   testcc(cc, dst);(*dst is cond. code*)
3147   (* test if the condition is satisfied *)
3148   if cc then begin(*33*)
3149     upperlow(src, length, 1, dst);(*dst + low bit*)
3150     (* finds the value of the LSB of the destination
3151        register assigning this value to the variable
3152        dst*)
3153     setindex(length, src);
3154     (* finds the correct number of the destination
3155        register. The order of the destination and
3156        source operands is inverse in this opcode*)
3157     rfsrcl(dst) := true;
3158     (* sets the LSB of the destination register to 1*)
3159     ic := ic+1;
3160
3161
3162
3163
3164
3165
3166
3167

```





# PROCEDURE LOGICAL.

The procedure executes the following instructions :

## FUNCTION

| operation | 1-2nd half<br>bytes | 3rd half<br>byte | 4th half<br>byte |
|-----------|---------------------|------------------|------------------|
| ORR R,R   | 132                 | R                | R                |
| OR R,R    | 133                 | R                | R                |
| ORR R,IM  | 04                  | = 0              | R                |
| OR R,IM   | 05                  | = 0              | R                |
| ORR R,IR  | 04                  | <> 0             | R                |
| OR R,IR   | 05                  | <> 0             | R                |
| ORR R,DA  | 68                  | = 0              | R                |
| OR R,DA   | 69                  | = 0              | R                |
| ORR R,X   | 68                  | <> 0             | R                |
| OR R,X    | 69                  | <> 0             | R                |
| XORR R,R  | 136                 | R                | R                |
| XOR R,R   | 137                 | R                | R                |
| XORR R,IM | 08                  | = 0              | R                |
| XOR R,IM  | 09                  | = 0              | R                |
| XORR R,IR | 08                  | <> 0             | R                |
| XOR R,IR  | 09                  | <> 0             | R                |
| XORR R,DA | 08                  | = 0              | R                |
| XOR R,DA  | 73                  | = 0              | R                |
| XORR R,X  | 72                  | <> 0             | R                |
| XOR R,X   | 73                  | <> 0             | R                |
| ANDR R,R  | 134                 | R                | R                |
| AND R,R   | 135                 | R                | R                |
| ANDR R,IM | 06                  | = 0              | R                |
| AND R,IM  | 07                  | = 0              | R                |
| ANDR R,IR | 07                  | <> 0             | R                |

```

3301
3302
3303
3304
3305      ANDB R,IR      06      <> 0      R
3306      ANDB R,DA      70      = 0      R
3307      AND  R,DA      71      = 0      R
3308      ANDB R,X      70      <> 0      R
3309      AND  R,X      71      <> 0      R
3310

```

The procedure executes all the opcodes of type UR,XUR ans AND as follows :

- selects the type of the operation setting the variable operation.
- selects one of the three base subcases in the case statement.
  - UR AND XUR R,R
    - calls the procedure ORANDXUR to perform the operation.
  - UR AND XUR R,IM or IR
    - retrieves the address
    - retrieves the data from the memory
    - sets the register R17 with the data
    - performs the operation between the Rdestination and the R17 calling the procedure ORANDXUR.
  - OR AND XUR R, DA or X
    - retrieves the address direct in the case of DA or calculates the combined address by calling the procedure DAORX.
    - retrieves the data
    - sets the register R17 with the data
    - performs the operation between the Rdestination and the R17 calling the procedure ORANDXUR.

#### PARAMFIFRS

-----: None.

#### GLOBAL VARIABLES

----- :- opcode is the decimal value of the opcode under execution.

----- ic is the decimal counter.Range 0..maxmem.

3364  
3365  
3366  
3367  
3368  
3369  
3370  
3371  
3372  
3373  
3374  
3375  
3376  
3377  
3378  
3379  
3380  
3381  
3382  
3383  
3384  
3385  
3386  
3387  
3388  
3389  
3390  
3391  
3392  
3393  
3394  
3395  
3396  
3397  
3398  
3399  
3400  
3401  
3402  
3403  
3404  
3405  
3406  
3407

- dst is the destination operand.Range 0..15.
- src is the source operand .Range 0..15.

# LOCAL VARIABLES

- :- l denotes the number of half bytes in the case of reference to the memory.Range 2 or 4.
- length denotes the length of the used register. Range 0..1.
- operation denotes the type of the operation. Range 1..3.
- val is used to find the value of the IM data in the case of IM and is used to find the address in the case of DA
- value is used to find the address in the case of IR.Range 0..maxmem.
- temp is the value of data that are retrieved from the memory.Range 0..maxinteq.

The procedure is called by : FXFLUIF.

The procedure calls : the procedures setlength,ORANDXOR, seteg, retvalreq.

\*\*\*\*\*)

procedure LOGICAL;

var

l, length, operation: integer;

val, value, temp: real;

begin(x1x)

setlength(l,length, l);

case opcode of

4, 5, 68, 69, 132, 133:

```

3421 operation := 1;(*OR*)
3422 6, 7, 70, 71, 134, 135:
3423 operation := 2;(*AND*)
3424 8, 9, 72, 73, 136, 137:
3425 operation := 3;(*XOR*)
3426 end;(*case*)
3427 case opcode of
3428 132, 133, 134, 135, 136, 137:
3429 begin(*Rdst ← Rdst (OR,AND,XOR) Rsrc*)(*4*)
3430 URANDXOR(dst, src, length, operation, true);
3431 ic := ic + 1
3432 end;(*4*)
3433 4, 5, 6, 7, 8, 9:
3434 begin(*Rdst ← Rdst (OP,AND,XOR) IM or IR*)(*5*)
3435 map(ic + 1, 1, 1, val);
3436 if src = 0 then begin(*6*)(*Rdst ← Rdst (OR,XOR,,AND) IM*)
3437 setreq(length, val, temporary);(*R17 ← val*)
3438 ic := ic + 3
3439 end else begin(*6*)(* IR*)(*7*)
3440 retvalreq(1, src, value);(*value ← [Rsrc]*)
3441 map(value, 1, 1, temp);(*temp ← mem[value]*)
3442 setreq(length, temp, temporary);(*R17 ← temp*)
3443 ic := ic + 1
3444 end;(*7*)
3445 URANDXOR(dst, temporary, length, operation, true)
3446 end;(*4*)
3447
3448 08, 09, 70, 71, 72, 73:
3449 begin(*Rdst ← Rdst (OR,AND,XOR) DA or X*)(*9*)
3450 DAORX(src, val);
3451 map(val, 1, 1, temp);(*temp ← mem[val+data]*)
3452 setreq(length, temp, temporary);(*R17 ← temp*)
3453 URANDXOR(dst, temporary, length, operation, true)
3454 end (* 9 *)
3455 end (*case*)
3456
3457
3458
3459
3460
3461
3462

```

end;(\*1 LOGICAL\*)

3484  
3485  
3486  
3487  
3488  
3489  
3490  
3491  
3492  
3493  
3494  
3495  
3496  
3497  
3498  
3499  
3500  
3501  
3502  
3503  
3504  
3505  
3506





|    |     |        |         |     |        |   |
|----|-----|--------|---------|-----|--------|---|
| 64 | 65  | CLR    | IR      | 13  | IR     | 8 |
|    | 66  | CLRB   | DA      | 76  | 0      | 8 |
|    | 67  | CLR    | DA      | 77  | 0      | 8 |
|    | 68  | CLRB   | X       | 76  | <> 0   | 8 |
|    | 69  | CLR    | X       | 77  | <> 0   | 8 |
|    | 70  | NUP    |         | 141 | 0      | 7 |
|    | 71  | NEGB   | R       | 140 | R      | 2 |
|    | 72  | NEG    | R       | 141 | R      | 2 |
|    | 73  | NEGB   | IR      | 12  | IR     | 2 |
|    | 74  | NEG    | IR      | 13  | IR     | 2 |
|    | 75  | NEGB   | DA      | 76  | 0      | 2 |
|    | 76  | NEG    | DA      | 77  | 0      | 2 |
|    | 77  | NEGB   | X       | 76  | <> 0   | 2 |
|    | 78  | NEG    | X       | 77  | <> 0   | 2 |
|    | 79  | CPR    | IR,IM   | 12  | IR     | 1 |
|    | 80  | CP     | IR,IM   | 13  | IR     | 1 |
|    | 81  | CPB    | DA,IM   | 76  | 0      | 1 |
|    | 82  | CP     | DA,IM   | 77  | 0      | 1 |
|    | 83  | CPB    | X,IM    | 76  | <> 0   | 1 |
|    | 84  | CP     | X,IM    | 77  | <> 0   | 1 |
|    | 85  | TSEIB  | R       | 140 | R      | 6 |
|    | 86  | TSEI   | P       | 141 | R      | 6 |
|    | 87  | TSFIR  | IR      | 12  | IR     | 6 |
|    | 88  | TSEI   | IR      | 13  | IR     | 6 |
|    | 89  | TSFIB  | DA      | 76  | DA     | 6 |
|    | 90  | TSEI   | DA      | 77  | 0      | 6 |
|    | 91  | TSEIB  | DA      | 76  | 0      | 6 |
|    | 92  | TSFIR  | X       | 76  | <> 0   | 6 |
|    | 93  | TSEI   | X       | 77  | <> 0   | 6 |
|    | 94  | PUSH   | IR,IM   | 13  | IR     | 9 |
|    | 95  | CUMFLG |         | 141 | NUMBER | 5 |
|    | 96  | SETFLG |         | 141 | NUMBER | 1 |
|    | 97  | RESFLG |         | 141 | NUMBER | 3 |
|    | 98  | LDCIIB | FLAGS,R | 140 | R      | 1 |
|    | 99  | LDCIIB | R,FLAGS | 140 | R      | 9 |
|    | 100 |        |         |     |        |   |
|    | 101 |        |         |     |        |   |
|    | 102 |        |         |     |        |   |
|    | 103 |        |         |     |        |   |
|    | 104 |        |         |     |        |   |
|    | 105 |        |         |     |        |   |
|    | 106 |        |         |     |        |   |
|    | 107 |        |         |     |        |   |



```

185 ----- : - dst is the destination operand.Range 0..15
186 - src is the source operand.Range 0..15.
187 - select is the source operand and it is used
188 to distinguish the different opcodes.Range
189 0..15.
190 - length denotes the length of the used register
191 Range 0..2.
192 - l denotes the number of the half bytes in the
193 case of reference to the memory.Range 2,4 or 8.
194 - x,val,value,temp as described in details in
195 the description of the operation.
196 - j is used as index in do loop during the
197 execution of the instruction LDM.Range 0..15.
198

```

199 The procedure is called by : EXECUTE.

200 The procedure calls : the procedures (NESTED) compl,compare,neq  
 201 tset,(NO NESTED) setlengthl,setsrclst,  
 202 retvalren,mao,setreg, ORANDXOR,setmem,  
 203 onetwocompl,upperlow,setindex.  
 204

205 \*\*\*\*\*)

206  
 207  
 208  
 209 procedure MULTIPER;

210 var

211 j, l, length, select, dst, src: integer;

212 x, val, value, temp: real;  
 213  
 214  
 215  
 216  
 217  
 218  
 219  
 220  
 221  
 222  
 223  
 224  
 225  
 226  
 227

```

241
242
243
244
245  (*****
246
247
248
249  FUNCTION
250  ----- The procedure performs one complement to the register
251           R17 and retrieves the contents of the R17 and sets the
252           destination memory to his new value.
253
254  PARAMETERS
255  ----- : by reference - temp denotes the address of the destina-
256           tion operand.Range 0..maxmem.
257
258  GLOBAL VARIABLES
259  ----- : (LOCAL IO MULTIPER )
260           - length denotes the length of the used register.
261           Range 0..2.
262           - value is the contents of the R17 after the opera-
263             tion.Range 0..maxinteg
264
265  LOCAL VARIABLES
266  -----: - the parameter.
267
268  The procedure is called by : MULTIOPER.
269  The procedure calls : the procedures onetwocompl,retvalreg,setmem.
270
271  *****
272
273
274
275  procedure compl(temp: real);
276
277  begin(*1*)
278      onetwocompl(temporary, length, 1, true);(*R17 ← not R17*)
279      retvalreg(length, temporary, value);(*value ← (R17)*)
280

```

```
304  
305     setmem(temp, 1, 1, value)  
306     end;(* 1 compl *)  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347
```



```

361
362
363
364
365 (*****
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400

```

PROCEDURE compare.

FUNCTION

----- The procedure performs the operation of compare between the destination and source operand as follows :

- the source operand is IM and retrieves his value assigning that to the variable value.
- sets the R18 with the data
- performs two complement to the register R18.
- adds the destination register R17 and the source data R18 affecting and the flags.
- sets or resets the C flag.

PARAMETER

----- : None.

GLOBAL VARIABLES

-----: (LOCAL TO MULTIOPER )

- value is the value of the IM data.Range 0..maxinteq.

LOCAL VARIABLES

----- :

- zero denotes if the data are equal to zero and the carry flag is not affected.Range 1 or F.

The procedure is called by : MULTIOPFK.

The procedure calls : the procedures map, setreg, onetwocompl, ADD.



```

481
482
483
484
485  (*****
486
487
488
489
490  FUNCTION
491  ----- The procedure changes the contents of the destination
492  operand to the opposite value as follows :
493  - performs two complement to the register R17
494  which contains the value of the destination
495  - assigns the contents of the R17 to the variable value
496  - sets the destination memory (parameter temp) to
497  the new value.
498
499  PARAMETER
500  -----: by value - temp denotes the address of the destination
501  operand.Range 0.. maxmem.
502
503  GLOBAL VARIABLES
504  ----- : (LOCAL TO THE MULTIOPER )
505  - length denotes the length of the used register.
506  Range 0..2.
507  - value is used to retrieve the contents of the
508  register R17.Range 0..maxinteg.
509
510  LOCAL VARIABLES
511  ----- : the parameter.
512
513  The procedure is called by : MULTOPER.
514  The procedure calls : the procedures onetwocompl,retvalreg,setmem.
515
516  (*****
517
518
519
520
521  procedure neg(temp: real);
522
523

```

```

545 begin(*1*)
546   onetwocompl(temporary, length, 2, true);(*R17 ← -R17*)
547   retvalreq(length, temporary, value);(*value ← R17*)
548   setmem(temp, 1, 1, value)
549   end;(* 1 neg *)
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587

```

```

601
602
603
604
605 (*****
606
607
608
609
610 FUNCTION
611 ----- The procedure performs the operation TSFI as follows :
612 - checks for one in the MSR of the destination
613   operand .
614 - sets or resets the S flag.
615 - sets the destination operand to all ones
616
617 PARAMETER
618 ----- : - by value - temp is the address of the destination
619   operand.Range 0..maxmem.
620
621 GLOBAL VARIABLES
622 ----- : (LOCAL TO MULTIOFPR)
623   - length denotes the length of the used register.
624   Range 0..1.
625
626 LOCAL VARIABLE
627 ----- : the parameter.
628
629 The procedure is called by : MULTIOFPR.
630 The procedure calls : the procedure setmem.
631
632 (*****
633
634
635
636
637 procedure tset(temp: real);
638
639 begin(*l*)
640   r(fcw)(s) := r(temporary)(7) and (length = 0)
641             or r(temporary)(15) and (length = 1 );
642
643 *****

```

```

665 setmem(temp, 1, 1, 65535)(* sets the memory to all ones*)
666 end;(*1 rset*)
667
668
669
670
671 begin(*1*)
672   setreg( 2 , 0 , temporary );
673   (* clears the two temporary registers *)
674   setsrclst(dst, select);
675   (* sets the variables dst and select. They are the 3 and 4th
676   half bytes of the opcode *)
677   serlenqth1(length, 1);
678   (* sets the variables length and 1 *)
679   if (opcode = 28) and (select = 0) then begin(*2*)
680     opcode := 13;(*IFST(L) IR, dst or 0*)
681     (* the execution of the opcode IEST and TESIL is the
682     same and it differs only to the value of length and 1 *)
683     select := 4
684   end;(*2*)
685   if (opcode = 92) and (select = 0) then begin(*3*)
686     opcode := 77;(*IFST(L) DA dst or 0*)
687     (* as above the execution is the same and it differs only
688     in the value of the variables length and 1 *)
689     select := 4
690   end;(*3*)
691   if (opcode = 156) and (select = 0) then begin(*4*)
692     opcode := 141;(*TEST (L), R dst or 0*)
693     (* the execution is the same and it differs in the value
694     of the variables length and 1 *)
695     select := 4
696   end;(*4*)
697   case opcode of(*TRUE in call statements means flags are affected*)
698     12, 13:
699       begin(*5*)
700
701
702
703
704
705
706
707

```



```

721      retvalreq(1, dst, temp);(*temp ← [Rdst]*)
722      (* assigns the contents of the destination register to
723       the variable temp *)
724
725      if (select <= 5) and (select <= 8) and (select <= 9)
726          then begin (*6*)
727
728              (* if select equal 5,8 or 9 then the destination is
729               not in the memory *)
730              map(temp, 1, 1, val);(*val ← mem[temp]*)
731              (* retrieves the contents of the destination
732               memory and assigns them to the variable val *)
733              setreg(lenqth, val, temporary)
734              (* sets the register R17 to the contents of the
735               memory in order to perform the operation *)
736              end; (* 6 *)
737
738      case select of
739      0:
740          compl(temp);(*COM (B) dst ← not dst*)
741          (* calls the procedure compl to perform the
742           operation *)
743
744      1:
745          (*CP (B) IR, dst ← IM*)
746          compare;
747          (* calls the nested procedure compare to perform
748           the operation *)
749
750      2:
751          (*NFG (R) dst ← - dst*)
752          neg(temp);
753          (* calls the procedure neg to perform the operation *)
754
755      4:
756          (*TEST (B,L) IR, dst ← dst or 0*)
757          ORANDXOR(temporary, 0, lenqth, 0, true);(*R17←R17 or 0*)
758          (* performs the OR operation calling the procedure
759           URANDXOR affecting the flags *)
760
761      5:

```

```

785 begin(*7*)(*STORE (B) IR,IM*)
786 map(ic + 1, 1, 1, value);(*value ← src*)
787 (* assigns the IM data to the variable value *)
788 setmem(temp, 1, 1, value);
789 (* sets the memory with the new IM data *)
790 end;(*7*)
791 6: (*TSFI(B) IR,S ← dst(msb)*)
792 tset(temp);
793 (* performs the operation calling the procedure
794 tset *)
795 8: (*CLR (B) IR, dst ← 0*)
796 setmem(temp, 1, 1, 0);(*mem[temp] ← 0*)
797 (* sets the memory to zero *)
798
799 9: begin(*PUSH IM*)(*8*)
800 temp := temp - 2;(*decrement the stackpointer*)
801 setreg(1, temp, dst);
802 (* sets the source register to his new value *)
803 map(ic + 1, 1, 4, value);(*value ← IM*)
804 (* retrieves the IM data assigning that to the
805 variable value *)
806 setmem(temp, 1, 4, value);(*PUSH*)
807 (* sets the memory to the IM data *)
808 end(* 8 *)
809 end (* case *);
810 if odd (select) then
811 ic := ic + 3
812 (* increments the ic by 3 because the opcode consists
813 from two words *)
814 else
815 ic := ic + 1;
816 (* the opcode consists from one word *)
817 end;(*5*)
818 140, 141:
819 begin(*10*)
820
821
822
823
824
825
826
827

```

```

841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882

```

```

case select of
0:
    onetwocompl(dst, length, 1, true);(*Rdst ← not Rdst*)
    (* performs the operation affecting and the flags *)

1:
    if opcode = 141 then begin(*11*)(*SETFLG flags(4:7)
    ← flags ( 4:7 ) or instruction *)
    for j := 0 to 3 do begin(*12*)
    (* the dst variable has the value of the opcode
    for the OR operation. This is performed with a
    mod operation starting from the PV flag which
    is the third bit of the FCW register *)
        r[fcw]pv + j := r[fcw]pv + j
        or (dst mod 2 = 1);

        dst := dst div 2
    end(*12*)
    end else begin(*11*)(*13 LDCtlb R, FLAGS *)
    (*Rdst (2:7) ← FLAGS (2:7)*)
    retvalren(length, fcw, val);(*val ← Flags (2:7)*)
    (* retrieves the contents of the low byte
    register of the FCW register and assigns that
    to the variable val *)
    setreg(length, val, dst)
    (* sets the destination register with the value
    of the FCW register *)
    end; (* 13 *)

2:
    begin(*NFG (B) R ← - R*)(*14*)
    onetwocompl(dst, length, 2, true);(*R ← - R*)
    (* performs two complement affecting and the flags *)
    retvalren(length, dst, value)
    (* retrieves the value of the register after
    the operation to check for overflow in the
    end of the procedure *)
    end;(*14*)

```

```

3, 5: begin(*15*) (*RESIFLG flags(4:7)+flags(4:7)
and not the dst operand of the instruction *)
(*CUMFLG flags(4:7) +flags(4:7) XOR instruction(4:7)*)
for j := 0 to 3 do begin(*16*)
(* the two opcodes differ to the last fourth
half byte which value has the variable select.
The operation is executed with the help of the
mod operation in order to find the ones and starts
from the PV flag which is the 3rd bit of the FCW
register *)
  if dst mod 2 = 1 then begin(*17*)
    if select = 3 then
      r[fcw]fpv + j] := false(*reset flag*)
    else
      r[fcw]fpv + j] := not r[fcw]fpv + j]
    end(*17*)(*complement the flags*);
    dst := dst div 2
  end(*16*)
end;(*15*)

4: ORANDXOR(dst, 0, length, 0, true);
(* TEST (B,L) Rdst ← Rdst or 0 *)
(* performs the operation affecting and the flags *)

6: begin(*18*) R ← dst(msb)(*18*)
uprflow(dst, length, 1, j);
(* finds the MSB of the destination register
assigning the value to the variable l *)
l := length;(*save length*)
(* saves the length of the register for latter use *)
temp := dst;(*saves dst*)
setindex(length, dst);
(* assigns the correct value of the dst register
to the variable dst *)

```

```

961   rlfcw[is] := rfdst[1];
962   (* copy the value of MSB to the S flag *)
963   setreg(j, 65535, trunc(temp))
964   (* sets the destination register to all ones *)
965   end;
966   (*18*)
967
968 7:
969   null;(*NOP*)
970   (* does not perform anything *)
971
972 R:
973   setreg(length, 0, dst)(*CLR (b) Rdst ←0*);
974   (* sets the destination register to zero *)
975
976 9:
977   begin (* 19 *)
978     (* LDCCLR flags, R FLAGS(2:7) ← Rsrc (2:7) *)
979     retvalreq(length, dst, value);(*value ← src (2:7) *)
980     (* retrieves the value of the source register
981        assigning the contents to the variable value *)
982     setreg(length, value, fcw);(*FLAGS (2:7) ← Rsrc (2:7)*)
983     (* sets the low half byte of the FCW register with
984        the source value *)
985     rlfcw[0] := false;
986     rlfcw[1] := false;
987     (* clears the two last bits of the FCW register *)
988     end(* 19 *)
989   end;(* case *)
990   ic := ic + 1
991 end;(*10*)
992 76, 77:
993   begin(*20*)
994     temp := 0;
995     map(ic + 1, 1, 4, val);(*val ← base addr *)
996     (* assigns the base address to the variable val *)
997     if dst <> 0 then(*then index*)
998       retvalreq(1, dst, temp);(*temp ←Rdisol*)
999       (* assigns the contents of the index register to the
1000

```

```

1024 variable temp *)
1025 val := val + temp;
1026 (* calculates the combined address either DA or X *)
1027 if (select <> 5) and (select <> 8) then begin(*21*)
1028   (* select = 0 COM (R) DA or X
1029     select = 1 CP (R) DA or X , dst - 1M
1030     select = 2 NEG (R) DA or X dst ← - dst
1031     select = 4 IFST(R) DA or X dst ← dst or 0
1032     select = 6 TSET(R) DA or X S ← dst(msb)
1033                                     *)
1034   map(val, 1, 1, value);(*value←memfval*)
1035   (* retrieves the contents of the destination memory
1036     and assigns his value to the variable value *)
1037   setreg(length, value, temporary)
1038   (* sets the temporary register R17 with the contents
1039     of the memory to perform the operation *)
1040   end; (* 21 *)
1041   case select of
1042     0:
1043       compl(val);(*COM (R) DA or X, dst ← not dst*)
1044       (* performs the operation calling the procedure
1045         compl *)
1046     1:
1047       (*CP(R) DA or X , dst -1M*)
1048       compare;
1049       (* performs the operation calling the procedure
1050         compare *)
1051     2:
1052       (*NEG (R) DA or X ,dst ← - dst*)
1053       neg(val);
1054       (* performs the operation calling the procedure
1055         neg *)
1056     4:
1057       (*IFST(R) DA or X, dst ← dst or 0*)
1058       ORANDXOR(temporary, 0, length, 0, true);

```



```

1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122

(* performs the operation affecting the flags *)
5:
    begin(*22*)(*STORE (B) DA or X, IM*)
    map(ic + 3, 1, 1, value);(*value ← IM*)
    (*memfval1 ← value *)
    (* assigns the IM data to the variable value *)
    setmem(val, 1, 1, value)
    (* sets the memory with the IM data *)
    end;(*22*)

6:
    (*ISET(B) DA or X, S← dst(msb)*)
    tset(val);
    (* performs the operation calling the procedure
       tset *)

8:
    (* CLR (B) RA or X , dst ← 0 *)
    setmem(val, 1, 1, 0)
    (* stores zero to the memory clearing the contents *)
    end; (* case *)
    if odd (select) then(*select = 1 or select = 5*)
    ic := ic + 5
    else
    ic := ic + 3
    end;(*20*)
2R, 92:
begin (* 50 *)
(* operation opcode 3rd 4th 5th 6th 7th 8th
   HALF BYTES
   LDM Rdst , Rdst + num -1 ← src
   src
   -----
   dst
   -----
   num
   num
   num
   dst
   src

   IR 2R 0 1 0 0 0 num
   DA 92 0 1 0 0 0 num
   X 92 <> 0 1 0 0 0 num
   STORE dst ← Rsrc , Rsrc + num - 1
   dst
   src

```

|      | IR | 28 | IR | 9 | 0 | K | 0 | num |
|------|----|----|----|---|---|---|---|-----|
| 1144 | DA | 92 | 0  | 9 | 0 | R | 0 | num |
| 1145 | X  | 92 | <> | 0 | 9 | R | 0 | num |
| 1146 |    |    |    |   |   |   |   | num |
| 1147 |    |    |    |   |   |   |   | *   |
| 1148 |    |    |    |   |   |   |   |     |
| 1149 |    |    |    |   |   |   |   |     |
| 1150 |    |    |    |   |   |   |   |     |
| 1151 |    |    |    |   |   |   |   |     |
| 1152 |    |    |    |   |   |   |   |     |
| 1153 |    |    |    |   |   |   |   |     |
| 1154 |    |    |    |   |   |   |   |     |
| 1155 |    |    |    |   |   |   |   |     |
| 1156 |    |    |    |   |   |   |   |     |
| 1157 |    |    |    |   |   |   |   |     |
| 1158 |    |    |    |   |   |   |   |     |
| 1159 |    |    |    |   |   |   |   |     |
| 1160 |    |    |    |   |   |   |   |     |
| 1161 |    |    |    |   |   |   |   |     |
| 1162 |    |    |    |   |   |   |   |     |
| 1163 |    |    |    |   |   |   |   |     |
| 1164 |    |    |    |   |   |   |   |     |
| 1165 |    |    |    |   |   |   |   |     |
| 1166 |    |    |    |   |   |   |   |     |
| 1167 |    |    |    |   |   |   |   |     |
| 1168 |    |    |    |   |   |   |   |     |
| 1169 |    |    |    |   |   |   |   |     |
| 1170 |    |    |    |   |   |   |   |     |
| 1171 |    |    |    |   |   |   |   |     |
| 1172 |    |    |    |   |   |   |   |     |
| 1173 |    |    |    |   |   |   |   |     |
| 1174 |    |    |    |   |   |   |   |     |
| 1175 |    |    |    |   |   |   |   |     |
| 1176 |    |    |    |   |   |   |   |     |
| 1177 |    |    |    |   |   |   |   |     |
| 1178 |    |    |    |   |   |   |   |     |
| 1179 |    |    |    |   |   |   |   |     |
| 1180 |    |    |    |   |   |   |   |     |
| 1181 |    |    |    |   |   |   |   |     |
| 1182 |    |    |    |   |   |   |   |     |
| 1183 |    |    |    |   |   |   |   |     |
| 1184 |    |    |    |   |   |   |   |     |
| 1185 |    |    |    |   |   |   |   |     |
| 1186 |    |    |    |   |   |   |   |     |
| 1187 |    |    |    |   |   |   |   |     |

```

map(ic + 1, 1, x);(*src or dst*)
(* assigns the value of the source or the destination
   operand to the variable x *)
src := trunc(x);
map(ic + 2, 1, 1, temp);(*temp + num*)
(* assigns the value of the number of repetition to
   the variable temp *)
if dst <> 0 then(*IR or X*)
  retvalreq(1, dst, value);(*value + (IR or X)*)
(* assigns the contents of the IR or X register
   to the variable value *)
if opcode = 92 then begin(*51*)
  map(ic + 3, 1, 4, val);(*val + addr (DA or X)*)
(* assigns the base or the address to the variable
   val *)
  ic := ic + 2
end;(*51*)
ic := ic + 3;
if opcode = 92 then begin(*52*)
  if dst <> 0 then
    value := val + value
    (* calculates the combined address (X) *)
  else
    value := val
    (* the address is DA *)
end;(*52*)
src := src + trunc(temp) - 1;
(* calculates the final register *)
if src > 15 then
  src := 15;

```

```

1201
1202
1203
1204
1205 (* in order to prevent overflow to the registers
1206 reduces the calculated final number of register
1207 to 15 *)
1208 case select of
1209 1:
1210     (*LDM*)
1211     for j := trunc(x) to src do begin(*53*)
1212         map(value, 1, 4, val);(* val ← contents of memory*)
1213         setreg(1, val, j);
1214         (* performs the operation retrieving the contents
1215         of the memory, assigning that to the variable
1216         val and then setting the register. After each
1217         operation increases the number of the register
1218         and the initial address *)
1219         value := value + 2
1220     end;(*53*)
1221
1222 9:
1223     (*STORE MULT*)
1224     for j := trunc(x) to src do begin(*54*)
1225         retvalreg(1, j, val);(*val ← register*)
1226         setmem(value, 1, 4, val);
1227         (* performs the operation retrieving the contents
1228         of the register and assigning his value to the
1229         variable val, then sets the memory to the new
1230         contents. After each operation increases the number
1231         of the register and the initial base address *)
1232         value := value + 2
1233     end;(*54*)
1234
1235     end; (* case *)
1236
1237     end(*50*)
1238
1239     end;(* case *)
1240     if select = 2 then begin (* 50 *) (* CASE OF NEG(R) *)
1241         r[fcw]lrvl := (length = 0) and (value = 128)
1242         or (length = 1) and (value = 32768);
1243         r[fcw]fc1 := r[fcw]lrvl = false;
1244

```

```
1264  
1265 end; (* 50 *)  
1266     (* because it is cleared if the result is zero otherwise  
1267     it indicates a borrow *)  
1268 end;(* 1 MULTIPLIER *)  
1269  
1270  
1271  
1272  
1273  
1274  
1275  
1276  
1277  
1278  
1279  
1280  
1281  
1282  
1283  
1284  
1285  
1286  
1287  
1288  
1289  
1290  
1291  
1292  
1293  
1294  
1295  
1296  
1297  
1298  
1299  
1300  
1301  
1302  
1303  
1304  
1305  
1306  
1307
```

1321

1322

1323

1324

1325 (\*\*\*\*\*)

1326

1327

1328

1329 FUNCTION

1330 -----The procedure executes the following instructions :

1331

| operation | 1-2nd half<br>bytes | 3rd half<br>byte | 4th half<br>byte |
|-----------|---------------------|------------------|------------------|
| EXB R,R   | 172                 | R                | R                |
| EX K,R    | 173                 | R                | R                |
| EXB R,IR  | 44                  | IR               | K                |
| EX R,IR   | 45                  | IR               | R                |
| EXB R,DA  | 108                 | 0                | R                |
| EX R,DA   | 109                 | 0                | R                |
| EXB R,X   | 108                 | <> 0             | R                |
| EX R,X    | 109                 | <> 0             | R                |

1334

1335

1336

1337

1338

1339

1340

1341

1342

1343

The procedure performs the followings :

- sets the variables lenqth,l.

- assigns the contents of the destination register to variable value.

- selects one main subcase as follows :

EX (B) R,R

1348

1349

1350

1351

1352

1353

1354

1355

1356

1357

1358

1359

1360

- assigns the contents of the source register to the variable val

- sets the two registers to their new values

EX (B) R,IR

- retrieves the contents of the source register to the variable tempo

EX (B) R, DA or X

```

1384 - assigns the source address to the variable temp.
1385 - if the opcode is not R,K then sets the memory and the register
1386 to their new values.
1387
1388
1389 PARAMETERS
1390 ----- :None.
1391
1392 GLOBAL VARIABLES
1393 -----
1394
1395 - ic is the current value of the decimal counter.
1396   Range 0..maxmem.
1397 - opcode is the decimal value of the opcode under
1398   execution.
1399 - dst is the destination operand .Range 0..15.
1400 - src is the source operand.Range 0..15.
1401
1402 LOCAL VARIABLES
1403 ----- : - length denotes the length of the used register.
1404           Range 0..1.
1405 - l denotes the number of half bytes in the case of
1406   reference to the memory.
1407 - temp,value,val as has been described in the detail
1408   operation of the procedure.
1409
1410 The procedure is called by : EXECUTE.
1411 The procedure calls : the procedures setlengthl,retvalreg,setreg,
1412   DAORX, setmem.
1413 *****
1414
1415 procedure EX;
1416
1417 var
1418   val, value, temp: real;
1419   length, l: integer;
1420
1421
1422
1423
1424
1425
1426
1427

```



```

1441 begin(*1*)
1442   setlength1(length, 1);
1443   retvalreg(length, dst, value);(*value ← [Rdst]**)
1444   case opcode of
1445     172, 173:
1446       begin(*2*)(*EX (B) R, R*)
1447         retvalreg(length, src, val);(*val ← [Rsrc]**)
1448         setreq(length, val, dst);(*Rsrc ← Rdst*)
1449         setreq(length, value, src);(*Rdst ← Rsrc*)
1450         ic := ic + 1
1451       end;(*2*)
1452     44, 45:
1453       begin(*3*)(*EX (B) R, IR*)
1454         retvalreg(1, src, temp);(*temp ← [Rsrc]**)
1455         ic := ic + 1
1456       end;(*3*)
1457     108, 109:
1458       begin(*4*)(*EX (B) R, DA or X*)
1459         DAorX(src, val);
1460         temp := val
1461       end(* 4 *)
1462     end; (*case*)
1463   if (opcode <> 172) and (opcode <> 173) then
1464     map(temp, 1, 1, val);(*val ← memory[temp]**)
1465     setmem(temp, 1, 1, value)(*mem ← value*);
1466     setreq(length, val, dst)(*Rdst ← value*)
1467   end;(* 1 FX *)

```

1504 (\*\*\*\*\*  
1505 \*\*\*\*\*  
1506 \*\*\*\*\*  
1507 \*\*\*\*\*  
1508 \*\*\*\*\*  
1509 \*\*\*\*\*  
1510 \*\*\*\*\*  
1511 \*\*\*\*\*  
1512 \*\*\*\*\*  
1513 \*\*\*\*\*  
1514 \*\*\*\*\*  
1515 \*\*\*\*\*  
1516 \*\*\*\*\*  
1517 \*\*\*\*\*  
1518 \*\*\*\*\*  
1519 \*\*\*\*\*  
1520 \*\*\*\*\*  
1521 \*\*\*\*\*  
1522 \*\*\*\*\*  
1523 \*\*\*\*\*  
1524 \*\*\*\*\*  
1525 \*\*\*\*\*  
1526 \*\*\*\*\*  
1527 \*\*\*\*\*  
1528 \*\*\*\*\*  
1529 \*\*\*\*\*  
1530 \*\*\*\*\*  
1531 \*\*\*\*\*  
1532 \*\*\*\*\*  
1533 \*\*\*\*\*  
1534 \*\*\*\*\*  
1535 \*\*\*\*\*  
1536 \*\*\*\*\*  
1537 \*\*\*\*\*  
1538 \*\*\*\*\*  
1539 \*\*\*\*\*  
1540 \*\*\*\*\*  
1541 \*\*\*\*\*  
1542 \*\*\*\*\*  
1543 \*\*\*\*\*  
1544 \*\*\*\*\*  
1545 \*\*\*\*\*  
1546 \*\*\*\*\*  
1547 \*\*\*\*\*

PROCEDURE LOAD.

FUNCTION

-----The procedure executes the following instructions :

| operation | 1-2nd half<br>bytes | 3rd half<br>byte | 4th half<br>byte |
|-----------|---------------------|------------------|------------------|
| -----     | -----               | -----            | -----            |
| LDB R,IM  | 32                  | 0                | R                |
| LD R,IM   | 33                  | 0                | R                |
| LDL R,IM  | 20                  | 0                | R                |
| LDB R,IR  | 32                  | <> 0             | R                |
| LD R,IR   | 33                  | <> 0             | R                |
| LDL R,IR  | 20                  |                  |                  |
| LDL R,IR  | 20                  | <> 0             | R                |
| LDB R,DA  | 96                  | 0                | R                |
| LD R,DA   | 97                  | 0                | R                |
| LDL R,DA  | 84                  | 0                | R                |
| LDB R,X   | 96                  | <> 0             | R                |
| LD R,X    | 97                  | <> 0             | R                |
| LDL R,X   | 84                  | <> 0             | R                |
| LDB R,R   | 160                 | R                | R                |
| LD R,R    | 161                 | R                | R                |
| LDL R,R   | 148                 | R                | R                |
| LDB R,BX  | 112                 | <> 0             | R                |
| LD R,BX   | 113                 | <> 0             | R                |
| LDL R,BX  | 117                 | <> 0             | R                |
| LDB R,DA  | 48                  | <> 0             | R                |
| LD R,DA   | 49                  | <> 0             | R                |
| LDL R,DA  | 53                  | <> 0             | R                |
| LDB R,IM  | 192                 | IMMEDIATE DATA   | IM               |
| LDB R,IM  | 189                 | R                | R                |
| LDA R,DA  | 118                 | 0                |                  |

```

1561
1562
1563
1564
1565 LDA R,X      118      <> 0      R
1566 LDA R,BA      52      <> 0      R
1567 LDA R,BX      116      <> 0      R
1568 LDAP R,RA      52      0      R
1569 LDR R,RA      48      0      R
1570 LDR R,RA      49      0      R
1571 POP R,IR      151      <> 0      R
1572 POPL R,IR     149      <> 0      R
1573 POP IR,IR      23      <> 0      IR
1574 POPL IR,IR     21      <> 0      IR
1575 POP IR,DA      87      IR      0
1576 POPL IR,DA     85      IR      0
1577 POP IR,X       87      <> 0      <> 0
1578 POPL IR,X      85      <> 0      <> 0
1579 LDR R,RA       53      0      R
1580

```

The details of the variables and what the procedure does is described in each different subcase.

PARAMETERS  
----- : None.

GLOBAL VARIABLES

```

----- : - opcode is the decimal value of the opcode
           under execution.
           - ic is the decimal value of the counter.
             Range 0..maxmem.
           - dst is the destination operand of the opcode
             Range 0..15.
           - src is the source operand of the opcode.
             Range 0..15.

```

LOCAL VARIABLES

```

----- : - 1 denotes the number of the half bytes in the

```

```

1624 case of reference to the memory.Range 2,4 or 8.
1625 - length denotes the length of the register.Range
1626 0..2.
1627 - val, value, x1,temp as described in each case
1628 in the description of the operations.
1629
1630

```

The procedure is called by : EXECUIF.

The procedure calls : the procedures IMorIR, setreg, DAorX, map,  
retvalreq, setmem.

```

1635 *****
1636 *****
1637 *****
1638 *****
1639 *****
1640 *****
1641 *****
1642 *****
1643 *****
1644 *****
1645 *****
1646 *****
1647 *****
1648 *****
1649 *****
1650 *****
1651 *****
1652 *****
1653 *****
1654 *****
1655 *****
1656 *****
1657 *****
1658 *****
1659 *****
1660 *****
1661 *****
1662 *****
1663 *****
1664 *****
1665 *****
1666 *****
1667 *****

```

```

procedure LOAD;

```

```

var

```

```

l, length: integer;

```

```

value, val, x1, temp: real;

```

```

begin(**)

```

```

setlength(l,length, l);

```

```

case opcode of

```

```

  20, 32, 33:

```

```

    begin(**)
      IMorIR(R,L) R, IM or IR*)(*2*)

```

```

      (* assigns the IM or IR data to the variable val *)

```

```

      setreg(length, val, dst)(* Rost + val *)

```

```

      (* sets the destination register to the decimal value
        of the data *)

```

```

    end; (* 2 *)

```

```

1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723

```

```

    96, 97, 84, 118:
begin(*3*)
    (*LD (B,L) or LDA ADDR R,DA or X*)
    DAorX (src, val);
    (* assigns the address of the source to the variable val*)
    if opcode <> 118 then
    (* if then opcode is 118 the it loads only address
    and so it does not retrieves data *)
    map(val, 1, 1, value)(*value ← memlval*)
    (* assigns the value of the source data to the variable
    value *)
    else
    value := val;(*LDA ADDR*)
    (* sets the used variable to set the destination
    register to the source address *)
    setren(length, value, dst)(*Rdst ← value *)
    (* sets the destination register to new data or
    address depending on the type of the opcode *)
    end; (* 3 *)
160, 161, 148:
begin(* 4 *)(*LD (B,L) R,R *)
    retvalreq(length, src, value);(*value ← [Rsrc]*)
    (* retrieves the contents of the source register and
    assigns his value to variable value *)
    setren(length, value, dst);(*Rdst ← value*)
    (* sets the destination register to his new value *)
    ic := ic + 1
end;(* 4 *)

192:
begin(* 5 *)(*LDB R,IM*)
    map(ic - 1, 1, 1, x1);
    (* the destination operand is in the second half
    byte of the opcode and assigns his value to the
    variable x1 *)

```

```

1744 dst := trunc(x1);
1745 map(ic, 1, 2, x1);
1746 (* the data is IM and they are part of the opcode,
1747    assigns the IM data to the variable x1 *)
1748 src := trunc(x1);
1749 setreq(0, src, dst);(*Rdst ← src*)
1750 (* sets the destination register to the IM data *)
1751 ic := ic + 1
1752 end;(* 5 *)
1753 48, 49, 53, 52:
1754 begin(* 6 *)(*LD (B,L) or LDA or LDAR R,BA*)
1755    (*or LDR(B,L),the assembler calculates the relative address *)
1756    DAORX (src,val);
1757    (* assigns the source address to the variable val *)
1758    if opcode < 52 then
1759      (* the opcode 52 denotes that address has to be loaded
1760         to the destination register *)
1761      map(val, 1, 1, value)
1762      (* assigns the source data from the memory to variable
1763         value *)
1764    else
1765      value := val;(*LDA or LDAR*)
1766      (* assigns the value of the address to the variable
1767         value which is used to set the destination register *)
1768      setren(length, value, dst)
1769      (* sets the destination register *)
1770    end;(* 6 *)
1771 112, 113, 117, 116:
1772 begin(* 7 *)(*LD (B,L) or LDA R,BX*)
1773    map(ic + 1, 1, 1, val);(*val ← Rindex*)
1774    (* assigns the number of the index register
1775       to the variable val *)
1776    retvalreg(1, trunc(val), value);(*value ← [Rindex]*)
1777    (* assigns the contents of the index register to
1778       variable value *)
1779 1780
1781
1782
1783
1784
1785
1786
1787

```



```

1801      retvalreg(1, src, val);(*val ← [Rbase]*)
1802      (* assigns the contents of the source (base) register
1803       to the variable val *)
1804      val := val + value;
1805      (* calculates the combined address *)
1806      if opcode <> 116 then
1807      (* opcode 116 denotes that only address will be
1808       loaded to the destination register *)
1809      map(val, 1, 1, value)(*value ← mem[val]*)
1810      (* retrieves the contents of the source data from
1811       the memory and assigns that to the variable value*)
1812      else
1813      value := val;(*LDA*)
1814      setreg(length, value, dst);(*Rdst ← val*)
1815      (* sets the destination register to the new address
1816       or the new data depending on the type of the opcode *)
1817      ic := ic + 3
1818      end;(* 7 *)
1819      189:
1820      begin(* 8 *)(*LDA R, IM*)
1821      setreg(length, dst, src);(*Rdst ← src*)
1822      (* the source operand is the IM data and sets the
1823       destination register to the value of the source *)
1824      ic := ic + 1
1825      end;(* 8 *)
1826      151, 149, 21, 23, 85, 87:
1827      begin(*PUP(l) R or IR or DA or X*)(* 9 *)
1828      retvalreg(1, src, val);(*val ← [Rsrc]*)
1829      (* assigns the contents of the source register
1830       (address) to the variable val *)
1831      map(val, 1, 1, temp);(*temp ← mem[val]*)
1832      (* retrieves the contents of the address from
1833       the memory and assigns the value to the variable temp *)
1834      val := val + 1 div 2;(*src ← src + 2 or + 4*)
1835      1840:
1836      1841:
1837      1842:

```

```

(* increments the original address by 2 or by 4
depending on the type of the opcode POP or PPOP*)
setreg(l, val, src);(*Rsrc ← Rsrc+2 or + 4*)
(* sets the source register to the new correct address*)
case opcode of
149, 151:
    begin (* 10 *)(* POP (L) R, IR *)
        setreg(length, temp, dst);(*R*)
        (* sets the destination register to the popped
        data *)
        ic := ic + 1
        end;(* 10 *)
21, 23:
    begin(*11*)(* POP (L) IR, IR *)
        retvalreg(l, dst, value);(*IR*)
        (* assigns the destination address to the variable
        value *)
        setmem(value, l, l, temp);
        (* sets the destination memory to the popped
        data *)
        ic := ic + 1
        end;(*11*)
85, 87:
    begin(*POP (L) DA or X, IR*) (* 12 *)
        DAorX(dst, value);
        (* assigns the destination address either DA
        or X to the variable value *)
        setmem(value, l, l, temp)
        (* sets the destination memory to the popped data *)
        end;(* 12 *)
        end(* case *)
        end(* 9 *)
        end (* case *)
        end;(* 1 LOAD *)

```



```

2045 The details of the use of the variables and what the procedure
2046 does is described in each subcase.
2047
2048
2049 PARAMETERS
2050 ----- : None.
2051
2052 GLOBAL VARIABLES
2053 ----- : - opcode is the decimal value of the opcode under
2054           execution.
2055           - dst is the destination operand.Range 0..15.
2056           - src is the source operand.Range 0..15.
2057           - ic is the decimal value of the counter.Range
2058             0..maxmem.
2059
2060 LOCAL VARIABLES
2061 ----- : - l denotes how many half bytes must be used
2062           in the case of reference to the memory.Range
2063             2,4,8.
2064           - length denotes the size of the used register.
2065             Range 0..2.
2066           - value is the decimal value of the data for
2067             storage.Range 0..maxinteq.
2068           - xl,val, temp are used for the calculation of
2069             the address.Range 0..maxmem.
2070
2071 The procedure is called by : FXCHIE.
2072 The procedure calls the procedures : setsrcdst, setlengthl, retvalreg,
2073                                     DAorX,map,setmem,setreg.
2074
2075 *****
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
           procedure STURE;

```

```

2101
2102
2103
2104
2105
2106
2107   l, length, src, dst: integer;
2108   temp, value, val, x1: real;
2109
2110
2111   begin(*1*)
2112       (* sets the dst and src operands *)
2113       setsrcdst(dst, src);
2114       (* sets the variables length, l *)
2115       setlengthl(length, l);
2116       (* retrieves the contents of the source register *)
2117       (* in all the cases except the PUSL because the
2118       variable length affects the source register *)
2119       if ( opcode <> 81 ) and ( opcode <> 17 ) then
2120           retvalreq(length, src, value);(*value ← [Rsrc]*)
2121       case opcode of
2122           50,51,55 : (* LD(R,L) BA,R   LDR (B,L) RA,R *)
2123               (*the assembler calculates the update address*)
2124               (* retrieves the destination address *)
2125               DAorλ(dst, val);
2126           114, 115, 119:
2127               begin(*5*)(*LD (B,L) BX,R*)
2128                   (* retrieves the base destination address *)
2129                   retvalreg(l, dst, val);(*val ← [Rdst]*)
2130                   (* finds the number of the index register *)
2131                   map(ic + l, l, 1, x1);(*src ← Rindex*)
2132                   (* changes the value to integer *)
2133                   src := trunc(x1);
2134                   (* retrieves the contents of the index register *)
2135                   retvalreg(l, src, temp);(*temp ← [Rindex]*)
2136                   val := val + temp;
2137                   (* calculates the final address *)
2138                   ic := ic + 3
2139                   end;(*5*)
2140
2141

```

```

2165 46, 47, 29:
2166 begin(*6*)(*LD (B,L) IR,R*)
2167 (* retrieves the destination address *)
2168 retvalreg(l, dst, val)(*val ← [Rdst]*);
2169 ic := ic + 1
2170 end;(*6*)
2171 110, 111, 93:
2172 begin(*7*)(*LD (B,L) DA or X, R*)
2173 (* retrieves the destination address *)
2174 DAddr(dst, val)
2175 end;
2176 147, 145, 19, 17, 83, 81:
2177 begin(*PUSH(L) DA or X or IR or R*)(*R*)
2178 (* retrieves the stackpointer *)
2179 retvalreg(l, dst, val)(*val ← [Rdst]*);
2180 (* decreases the stackpointer depending on the type
2181 of the opcode *)
2182 val := val - 1 div 2;(*dst ← dst -2 or -4*)
2183 (* sets the used register as stackpointer to his new value*)
2184 setreg(l, val, dst);
2185 case opcode of
2186 147, 145: (* PUSH R,R *)
2187   ic := ic + 1;(*R*)
2188   (* the value for push has been assigned to the vari-
2189     ble value *)
2190 19, 17: (* PUSH IR,R *)
2191   begin(*9*)
2192     (* finds the address of the source *)
2193     retvalreg(l, src, temp);(*temp ← [Rsrc]*)(*[R*)
2194     (* retrieves the contents of the address *)
2195     map(temp, l, l, value);
2196     (*value ← mem[temp]*)
2197     ic := ic + 1
2198   end; (* 9 *)
2199 81, 83: (* PUSH (l) IR, DA or X *)
2200
2201
2202
2203
2204
2205
2206
2207

```



```

2221
2222
2223
2224
2225
2226      begin(*10*)
2227      (* finds the address of the source*)
2228      UAddr(src, temp);(*temp ← address*)
2229      (* retrieves the contents of the source *)
2230      map(temp, 1, 1, value)
2231      end(* 10 *)
2232      end(*case*)
2233      end(* 8 *)
2234      end; (* case *)
2235      (* sets the memory (PUSH ) with address = val and data = value*)
2236      setmem(val, 1, 1, value)
2237      end; (* 1 STORE *)
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257

```

```
4  
5 program simulator(input, output);  
6 #include 'firstpart.i'  
7 #include 'secondpart.i'  
8 #include 'thirdpart.i'  
9 #include 'fourthpart.i'
```

```
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47
```

```

61
62
63
64
65 (*****
66
67         PROCEDURE er.
68 The procedure is called by : displaymix,FILL,JUMP.
69
70 *****
71
72
73 procedure er;
74
75
76
77 begin (*1*)
78     writeln('      ** Odd address encountered **');
79 end; (*1*)
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

```

125 (*****
126
127         PROCEDURE boot.
128
129 FUNCTION
130 ----- : Initializes the global variables, fills the memory with
131 zeros, sets the two stackpointers in the default values
132 (system stackpointer at 0y00 and the normal stackpointer
133 at the maximum memory), sets the RFC register to 0, loads
134 the PSA area from a file and sets the PSAUFF register
135 to the base address of the PSA area.
136
137 PARAMETERS
138 ----- : None.
139
140 GLOBAL VARIABLES
141 ----- Initializes all the global variables.
142
143 LOCAL VARIABLES
144 ----- :
145         - byte is used as half byte counter at the memory
146           initialization. Range 0..1.
147         - i is used as index in the loops for the initiali-
148           zation of the memory.
149         - ch is used to read characters from the file,
150           that contains the PSA area code and the subrou-
151           tine to handle the interrupts.
152         - psacode is the text file containing the code
153           for the PSA area. Also the user may includes
154           and the subroutines to service the TRAPS.
155
156 The procedure is called by : the main program.
157 The procedure calls : The procedure setreg.
158 *****
159
160
161
162
163
164
165
166
167

```



```

i:= psaarea; (* address of the PSA area *)
while not eof(psacode) do begin (*2*)
  if not eoln(psacode) then begin(*3*)
    read(psacode,ch);
    memoryfil(hytel := ch;
    hyte := (hytel+1) mod 2;
    if hyte = 0 then
      i := i+1;
    end(*3*)
  else
    readln(psacode);
  end;(*2*)
end; (* 1 hoot *)

```

245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287



```

301
302
303
304
305 (*****
306
307
308           PROCEDURE skipblanks.
309
310 FUNCTION
311 ----- : The procedure skips spaces until eoln is encountered
312           or character different than the blanks and returns
313           to the calling procedure the character.
314
315 PARAMETERS
316 ----- : by reference - ch is the returned character.Range
317           all the valid characters.
318
319 GLOBAL VARIABLES
320 ----- : None.
321
322 LOCAL VARIABLES
323 ----- : -The parameter 'ch'.
324
325 The procedure is called by : readaddress,DISPLAY,FILL,REG,MUNITOR.
326 The procedure calls : None.
327
328 (*****
329
330
331
332
333
334           procedure skipblanks(var ch: char);
335
336           begin (*1*)
337             repeat
338               if not eoln(input) then
339                 read(ch)
340             until eoln(input) or (ch <> ' ')(*skip blanks *)
341           end; (* 1 skipblanks *)

```

```

425 (*****
426
427         PROCEDURE load.
428
429 FUNCTION
430 ----- The procedure loads the object code in hexadecimal characters
431 in the memory. It checks for end of file condition and also
432 if the object code is greater than the maximum available,
433 memory minus the area for the PSA area.
434
435 PARAMETERS
436 ----- :None.
437
438 GLOBAL VARIABLES
439 ----- : - loprogram denotes that the object code has been
440 loaded, Range T or F.
441          - memory is the used array as real memory.
442
443 LOCAL VARIABLES
444 ----- : - bytes is used as half byte counter in the loading.
445          Range 0..1.
446          - index is used as memory index during the loading.
447          Range 0..PSA area.
448          - code is the object file.
449
450 The procedure is called by : MONITOR.
451 The procedure calls :None.
452
453 *****
454
455 procedure load:
456
457 var
458
459
460
461
462
463
464
465
466
467

```

```

byte, index: integer;
code: text;

begin (*1*)
  loprogram := true;
  reset(code, 'executecode'); (*ready for reading only*)
  index := 0; (* index of memory *)
  byte := 0;
  while not eof(code) and (index <= psaarea) do begin (*2*)
    read(code, ch); (* loads the program *)
    memory[index]{byte} := ch;
    byte := (byte + 1) mod 2; (* increases the byte index *)
    if byte = 0 then
      index := index + 1 (* increase the index of the memory *)
    end; (*2*)
  end; (* 1 load *)

```

# PROCEDURE dechex.

## FUNCTION

----- :The procedure is used to transform a decimal number to equivalent hex and then to print the result to the screen. For this reason uses an array of 4 characters which is initialized to spaces. Depending on the value of the parameter byte it prints four characters or two and two spaces depending if the value belongs to word or to byte.

## PARAMETERS

----- by reference - 'n' is the decimal value to be transformed to hex one. Range 0..maxinteg.  
 - byte denotes if the value belongs to a byte or to a word. Range I or F.

## GLOBAL VARIABLES

----- : None.

## LOCAL VARIABLES

----- : - j is used to control the transforming mechanism depending on the value of the parameter byte. Range 1 or 3 (byte or word).  
 - hexval is the used array.  
 - temp is the result of the mod operation of the remaining value divided by 16. Range 0..15.  
 - num is the quotient of the value divided by 16. Range 0..maxinteg.

The procedure calls : None.

The procedure is called by : REGISTER, EXECUTE, compare, displaymix, RFG.

```

601
602
603
604
605
606
607 *****
608
609
610 procedure dechex(n: integer;byte : boolean);
611
612 var
613   j, temp, num: integer;
614   hexval: packed array [0..3] of char;
615
616   begin (*2*)
617     if byte then
618       j := 1
619     else
620       j := 3;
621     hexval := ' ' (*fills the array with spaces*)
622     num := n; (* denotes the number to be trasformed from*)
623             (*decimal to hex value*)
624     while j >= 0 do begin (*2*)
625       temp := num mod 16;
626       if temp < 10 then
627         temp := temp + 48 (* diqit 0..9 *)
628       else
629         temp := temp + 55;(* character A..F *)
630       hexval[j] := chr(temp);(* fill the array *)
631       num := num div 16;
632       j := j - 1
633     end; (*2*)
634     write(hexval);(* write the word to the screen *)
635     end; (* 1 dechex *)
636
637
638
639
640

```

```

665 (*****
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707

PROCEDURE REGISTER.

FUNCTION
-----
The procedure prints the contents of the R0..R15 , RPC and
RFC REGISTERS in two lines.It calls successively the proce-
dures retvalreg to return the value of the register and the
dechex to print the value of the register in hex form.

PARAMETERS
----- None.

GLOBAL VARIABLES
----- None.

LOCAL VARIABLES
----- : - j is used as a register index .Range 0..19.
- val is the returned value of the register.
Range 0..maxinteq.

The procedure is called by : EXECUTE, REG.
The procedure calls : The procedures retvalreg , dechex.

*****
procedure REGISTER:
var
j: integer;
val: real;
begin (*1*)

```



```

721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762

if not eoln(inn) then
  readln;(* checks if exists end of line and if not *)
  (* clears the input buffer *)
  for j := 0 to 16 do begin (*2*)
    if j = 0 then begin (*3*)
      writeln(' R0---R1---R2---R3---R4---R5---R6---R7---',
        'R8---R9---R10---');
      write(' ') (* left margin *)
    end; (*3*)
    if j = 11 then begin (*4*)(* second line of registers *)
      writeln(' R11---R12---R13---R14---R15---RPC---RFC---');
      write(' ') (* left margin *)
    end; (*4*)
    retvalreq(1, j, val);(* value of the register *)
    dechex(trunc(val),false);(* prints the value of the register *)
    write(' ');(* separator between the values of the registers *)
    if j = 10 then (* skips line *)
      writeln
    end; (*2*)
    retvalreq(1, fcw, val);
    dechex(trunc(val),false);(* prints the value of the FCW register *)
    writeln
  end; (* 1 REGISTER *)

```

# PROCEDURE decode.

## FUNCTION

-----The procedure map the hex value of the first byte of the opcode to decimal value. There are conflicts in the mapping mechanism because there are opcodes that have only a half byte size to denote the operation. These are the following:

operation opcode

-----

|          |   |
|----------|---|
| LDR R, M | C |
| CALR     | D |
| DJNZ     | F |
| JR       | E |

In the above cases the hex character is multiplied by 16 and so the opcodes '0C' and '0F' have different values.

## PARAMETERS

----- by reference :- opcode is the returned decimal value of the opcode. Range 0..192, 208, 224, 240.  
by value :- line is the byte to be decoded.

## GLOBAL VARIABLES

----- : None.

## LOCAL VARIABLES

----- : - the parameter opcode .  
- k is the returned value of the hex character for decoding from the procedure valuechar. Range 0..15.  
- j is used as index in the memory .Range 0..1.

The procedure is called by : EXECUTE.

841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881

The procedure calls : the procedure valuechar.

\*\*\*\*\*)

procedure decode(var opcode: integer; line: alpha);

```
var
k, j: integer;
begin (*1*)
j := 0;
if line[j] in ['0'..'f'] then
    opcode := (ord(line[j]) - 55) * hexv
else begin (*2*)
    valuechar(line[j],k);
    opcode := hexv * k;
    valuechar(line[j + 1], k);
    opcode := opcode + k
end (*2*)
end; (*decode 1*)
```

905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947

## FUNCTION

The procedure executes the following functions in for ever loop and in order :

- calls the procedure decode to return the value of the opcode.
- checks if the TRACE mode exists .If yes then calls the procedure PEGSIEK to print the contents of all the registers and also decreases the counter next which denotes in how many instructions the TRACE mode will be executed.
- checks if the opcode is valid.If yes then calls the appropriate procedure.In the case that the opcode is not valid or the opcode is privileged one and the mode of operation is normal calls the procedure settrap to call the procedure IRAP.
- sets the counter register.
- checks if the counter produces overflow or underflow.
- checks if the address is odd.the feature is under comments because protects only the user program from software errors.
- checks if BREAKPOINT is fulfilled and if yes decreases the nthtime variable denoting how many times the BREAKPOINT must be encountered in order the user's program return to the MONITOR.

## PARAMETERS

.....: None.

## GLOBAL VARIABLES

```

-----: - IPACE denotes that the user has specified TRACE mode.
          Range 1 or F.
- next denotes how many instructions will be traced.
  Range 1..maxmem.

```

```

961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001

```

- opcode is the decimal value of the opcode.
- Range 0..192,208,224,240.
- ic is the decimal counter.Range 0..maxmem.
- alldone denotes if halt codition is true or
- BREAKPOINT has occurred.Range T or F.
- BREAKPOINT denotes the address for breakpoint.Range
- 100 or 0..maxmem.
- nthtime denotes how many times the BREAKPOINT must
- be encountered.Range 0..maxinteg
- src,dst are the decimal values of the source and
- destination operand.

#### LOCAL VARIABLES

----- : None.

The procedure calls :The procedures decode,REGISTER,setsrddst,settrap,  
EX,STORE,MULTIPLY,LOGICAL,ARITHMETIC, DIVMULT,  
CALLREF, BITMAN, INPUTOUTPUT,ROTATESHIFT,  
BLOCKTRANSFER,TRANSLATE, IREF,LDCTL,LPDS,DAR,  
setreq, error, dechex.

The procedure is called by : The main program.

\*\*\*\*\*)

procedure EXECUTE;

```

1025 (*****
1026
1027
1028
1029 PROCEDURE settrap
1030
1031 ----- Calls the procedure TRAP depending on the parameter
1032 and decreases the ic counter by one in order to be saved
1033 the correct value of the ic.
1034
1035 PARAMETER
1036 ----- by reference :- 'n' denotes the offset in the PSA area
1037 as follows :
1038 offset meaning
1039 -----
1040 0 RESERVED
1041 4 UNIMPLEMENTED INSTRUCTION.
1042 8 PRIVILEGED INSTRUCTION
1043 12 SYSTEM CALL
1044
1045 The procedure calls : The procedure IRAP.
1046 The procedure is called by : EXECUTE as nested procedure.
1047
1048 *****
1049
1050 procedure settrap(n : integer);
1051 begin(*1*)
1052   ic := ic-1;
1053   IRAP(n,opcode*256);
1054   end;(*1 settran *)
1055
1056
1057 begin (*1*)
1058   repeat
1059     decode(opcode, memory(ic));
1060
1061
1062
1063
1064
1065
1066
1067

```





1145  
1146  
1147  
1148  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1160  
1161  
1162  
1163  
1164  
1165  
1166  
1167  
1168  
1169  
1170  
1171  
1172  
1173  
1174  
1175  
1176  
1177  
1178  
1179  
1180  
1181  
1182  
1183  
1184  
1185  
1186  
1187

end ;(\*/\*\*)

44, 45, 108, 109, 172, 173:

EX;

17, 19, 29, 46, 47, 50, 51,  
55, 81, 83, 93, 110, 111, 114,  
115, 119, 145, 147:

STORE;

12, 13, 28, 76, 77, 92, 140,  
141, 156:

MULTIPLIER ;

4, 5, 6, 7, 8, 9, 68,  
69, 70, 71, 72, 73, 132,  
133, 134, 135, 136, 137:

LOGICAL;

0, 1, 2, 3, 10, 11, 16,  
18, 22, 40, 41, 42, 43, 64,  
65, 66, 67, 74, 75, 80, 82,  
86, 104, 105, 106, 107, 128, 129,  
130, 131, 138, 139, 144, 146, 150,  
168, 169, 170, 171, 174, 175, 177,  
180, 181, 182, 183:

ARITHMETIC;

24, 25, 26, 27, 88, 89, 90,  
91, 152, 153, 154, 155:

```

1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242

    DTVMULT;
        30, 31, 94, 95, 158, 208,
        224, 240:

    CALIRFI;
        34, 35, 36, 37, 38, 39, 98,
        99, 100, 101, 102, 103, 162,
        163, 164, 165, 166, 167:

    BITMAN;
        58, 59, 60, 61, 62, 63:
        if not r(fcw)[sn] then
            settrap(8)
        else
            INPUTOUTPUT;

        178, 179, 188, 190:

    ROTATESHIFT;

        186, 187:

    BLUCLTRANSFER;

        184:

        123:

        if (src = 0) and (dst = 0) then

```

```

begin(*S*)
  if not r(fcw){sn} then
    settran(8)
  else
    IRET
  end (* 5 *)
  else
    error(7);
    (*MRLIT ,MREQ, MRES, MSET*)
125:
  if not r(fcw){sn} then
    settrap(8)
  else
    LDCIL;
127:
  (*SYSTEM CALL*)
  if not r(fcw){sn} then
    settrap(12)
  else
    ic:= ic+1;
57, 121:
  if not r(fcw){sn} then
    settrap(8)
  else
    LPDS;
176:
    DAB;

```

```

1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307

```

```

1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
124:
    (*FI or DI*)
    if not rffcwl[sn] then
        settrap(8)
    else begin (*6*)
        rffcwl[12] := (dst = 4) or (dst = 5);
        rffcwl[11] := (dst = 6) or (dst = 4);
        ic := ic + 1;
        end;(*6*)
    end (*case opcode*)
else
    begin (*7*)
    error(8);(*not implemented instruction by the Z-8000*)
    settrap(4);
    end;(*7*)

    setrea(1, ic, counter);(*sets the counter*)
    if (ic < 0) or (ic > maxmem - 3) then
        error(1);(*out of memory range *)
    (*if odd(ic) then
        error(9);*)(*odd address encountered during the execution *)
    if (next <= 0) and TRACE then begin (*2a *)
        TRACE := false;(*if next counter becomes zero then the *)
        (* TRACE option becomes false *)
        alldone := true; (* terminates the user's program*)
    end; (*2a *)
    if ic = BREAKPOINT then begin(*10*)(* break point encountered*)
        nthtime := nthtime - 1; (*decreases the specified from the *)
        (* the user limit *)
        alldone := nthtime = 0;
    if alldone then begin (*8*)
        write('      ** message : BREAK AT ');
        dechex(ic,false);(* hex value of the counter *)

```

```
1385      writeLn
1386      end (* 9 *)
1387      end (* 8 *)
1388
1389      until allDone
1390      end; (* 1 EXECUTE *)
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
```



```

1441
1442
1443
1444
1445 (*****
1446
1447
1448
1449
1450 FUNCTION
1451 ----- The procedure reads an hexadecimal address from the screen
1452 and trasforms to decimal.Also it cheks if the address is
1453 in the bounds of the memory and if the typed by the user
1454 characters are valid.
1455
1456 PARAMETERS
1457 ----- by reference : -'n' is the decimal equivalent of the hexa-
1458 decimal address.Range 0..maxmem.
1459 - correct denotes if the address is correct
1460 and the typed characters are hex characters.
1461 Range 1 or F.
1462
1463 GLOBAL VARIABLES
1464 ----- : - validch is a set of the valid hex characters,
1465 ['0'..'9','A'..'F'].
1466
1467 LOCAL VARIABLES
1468 ----- : - The parameters.
1469 - ch is used to read characters from the screen.
1470 Range ASCII characters.
1471 - k is used to trasform the hex characters to deci-
1472 mal values.Range 0..F.
1473
1474 The procedure calls :The procedure valuechar.
1475 The procedure is called by : firstsecond,COMPARE ,JUMP,NEXT.
1476
1477 (*****
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000

```

1505  
1506  
1507  
1508  
1509  
1510  
1511  
1512  
1513  
1514  
1515  
1516  
1517  
1518  
1519  
1520  
1521  
1522  
1523  
1524  
1525  
1526  
1527  
1528  
1529  
1530  
1531  
1532  
1533  
1534  
1535  
1536  
1537  
1538  
1539  
1540  
1541  
1542  
1543  
1544  
1545  
1546  
1547

```

procedure readaddress(var n: integer; var correct: boolean);

var
  ch: char;
  k: integer;

begin (*1*)
  correct := true;
  n := 0;
  skipblanks(ch);
  if ch in validch then begin (*2*)
    valuechar(ch, n); (* transforms the hex character to decimal *)
    (* value *)
    if not eoln(input) then
      repeat
        read(ch);
        if ch in validch then begin (*3*)
          valuechar(ch, k);
          n := n * 16 + k (* in each step increases the base by 16 *)
        end else if ch <> ' ' then
          correct := false
        until not (ch in validch) or eoln(input)
          or ( n >= maxinteq)
        (* terminates if the character is not valid or the end of
        line is encountered or the address is greater than the maxinteq *)
      end else (*2*)
        correct := false;
    if not (correct and ((n <= maxmem) and (n >= 0))
      and ( ch in [' ', '0'..'9', 'A'..'F'] ) ) then
      correct := false;
    (* checks if the final result is correct *)
  end; (* 1 readaddress *)

```

```

1621
1622
1623
1624
1625 (*****
1626
1627 PROCEDURE firstsecond.
1628
1629 FUNCTION
1630 ----- The procedure is used to read two addresses from the screen
1631 if the first one is correct.The procedure is called from
1632 the procedures that execute commands of the MUNITUR with two
1633 addresses.
1634
1635 PARAMETERS
1636 ----- by reference: - first is the first address.Range 0..maxmem.
1637 - second is the second adress.Range 0..maxmem.
1638 - correct denotes if and the two addresses
1639 are correct .Range T or F.
1640
1641 GLOBAL VARIABLES
1642 ----- : None.
1643
1644 LOCAL VARIABLES
1645 ----- : - The parameters.
1646
1647 The procedure calls : the procedure readaddress.
1648 The procedure is called by : BREAK, COMPARE,DISPLAY,FILL,MOVE.
1649
1650
1651
1652 (*****
1653
1654
1655
1656 procedure firstsecond(var first,second : integer;
1657 var correct : boolean);
1658
1659
1660
1661

```

```
1685 begin (*1*)
1686   readaddress(first, correct);(*first address*)
1687   if correct then
1688     readaddress(second, correct); (*second address*)
1689   end; (*1*)
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
```

1741

1742

1743

1744

1745

1746

1747

1748

1749

1750 FUNCTION

1751 ----- The procedure executes the BREAK command of the MONITOR.

1752 ex. of command : B <address> <n>

1753 The BREAK command is used to set a breakpoint at a given

1754 even address, first clearing any breakpoint previously set.

1755 The number after the address denotes that the user program

1756 will not be interrupted until the nth time the breakpoint

1757 instruction is encountered. If the BREAK command is issued

1758 with no parameters it clears any previously set breakpoint.

1759 When user program is suspended by the break command the mo-

1760 nitor prints a message informing of the break and the address

1761 at which it occurred.

1762 ex. message BREAK AT 0A04

1763

1764 PARAMETERS

1765 ----- : by reference - correct denotes that the command is

1766 correct. Range T or F.

1767

1768 GLOBAL VARIABLES

1769 ----- - BREAKPOINT denotes the address for breakpoint.

1770 Range 0..maxmem or -100 (denotes no breakpoint).

1771 - all done denotes that the user program can start.

1772 Range T or F.

1773 - nthtime denotes how many times the BREAKPOINT must

1774 be encountered. Range 0..maxinteg.

1775 LOCAL VARIABLES

1776 -----: -The parameter.

1777

1778 The procedure calls : the procedure firstsecond.

1779 The procedure is called by : MONITOR.

1780

1781

1782

```

1805 *****
1806 *****
1807 *****
1808 *****
1809 *****
1810 *****
1811 *****
1812 *****
1813 *****
1814 *****
1815 *****
1816 *****
1817 *****
1818 *****
1819 *****
1820 *****
1821 *****
1822 *****
1823 *****
1824 *****
1825 *****
1826 *****
1827 *****
1828 *****
1829 *****
1830 *****
1831 *****
1832 *****
1833 *****
1834 *****
1835 *****
1836 *****
1837 *****
1838 *****
1839 *****
1840 *****
1841 *****
1842 *****
1843 *****
1844 *****
1845 *****
1846 *****
1847 *****

procedure BREAK(var correct: boolean);

var
  address, number: integer;

begin (*1*)
  firstsecond(address,number,correct);(* reads the first address*)
  (*and the number that the address must be encountered in order*)
  (* the break occurs *)
  if correct then begin(*2*)(*the address and the number are correct*)
    BREAKPOINT := address;
    nthtime := number;
    alldone := false(*false in order to start the simulator*)
  end else (*2a*)
    BREAKPOINT := -100 (*clear any previously set breakpoint*)
  end; (* 1 BREAK *)

```



```

1861
1862
1863
1864
1865 (*****
1866
1867
1868 ex of command : C < address 1 > < address 2 > < n >
1869 FUNCTION
1870 ----- The procedure executes the COMPARE command of the MONITOR.
1871 The COMPARE command is used to compare the contents of two
1872 blocks of the memory. <address1> and <address2> specify the
1873 starting addresses of the two blocks, while the <n> specifies
1874 the number of the bytes to be compared. If any location of the
1875 two blocks differ the addresses and the contents of those lo-
1876 catons are printed on the terminal.
1877 ex of command : C 0400 0500 30 < cr >.
1878 ex of answer if they differ :
1879 first address 040A 0B second address 050A 0C
1880
1881 PARAMETERS
1882 ----- : by reference - correct denotes that the command is correct.
1883 Range 1 or f.
1884
1885 GLOBAL VARIABLES
1886 ----- - memory is the used real memory.
1887
1888 LOCAL VARIABLES
1889 ----- : -The parameter.
1890 - first denotes the first address.Range 0..maxmem.
1891 - second denotes the second address.Range 0..maxmem.
1892 - times denotes how many times the comparisons will
1893 be executed.Range 0 .. maxmem.
1894 - j is used as offset from the base addresses for
1895 comparisons in the while loop.Range 0..times.
1896
1897 The procedure call : The procedures firstsecond,readaddress ,dechex.
1898 The procedure is called by : MONITOR.
1899
1900
1901
1902
1903

```

```

1925 *****
1926 *****
1927 *****
1928 *****
1929 *****
1930 *****
1931 *****
1932 *****
1933 *****
1934 *****
1935 *****
1936 *****
1937 *****
1938 *****
1939 *****
1940 *****
1941 *****
1942 *****
1943 *****
1944 *****
1945 *****
1946 *****
1947 *****
1948 *****
1949 *****
1950 *****
1951 *****
1952 *****
1953 *****
1954 *****
1955 *****
1956 *****
1957 *****
1958 *****
1959 *****
1960 *****
1961 *****
1962 *****
1963 *****
1964 *****
1965 *****
1966 *****
1967 *****

procedure COMPARE(var correct: boolean);

var
    j, first, second, times: integer;

begin (*1*)
    firstsecond(first,second,correct);
    (* reads the two base addresses *)
    if correct then (* correct the addresses *)
        readadress(times, correct); (* number of comparisons *)
    if correct then (* correct all the command *)
        while (first + j <= maxmem) and (second + j <= maxmem)
            and (j <= times) do begin (*3*)
                if memory[first + j] <> memory[second + j] then begin (*4*)
                    write('  * first address ');
                    dechex(first + j,false);(*write the first address*)
                    write(' ',memory[first + j], ' ');
                    (* prints the contents of the memory *)
                    write(' Second address ');
                    dechex(second + j,false); (* write the second address *)
                    writeln(' ',memory[second + j]);
                    (* prints the contents of the second address*)
                end; (*4*)
                j := j + 1
            end (*3 while *)
        end; (* 1 COMPARE *)

```



```

2105 that are typed by the user and it checks if they
2106 are valid hex characters.
2107 ex of command : < D 0500 B W < cr >
2108 or < D 0500 A <cr>
2109
2110 PARAMETERS
2111 ----- by reference - correct denotes that the command is correct.
2112 Range I or F.
2113
2114 GLOBAL VARIABLES
2115 -----: - validch is the set of the valid hex characters.
2116
2117 LOCAL VARIABLES
2118 -----: - The parameter.
2119 - max is the final address of the display operation.
2120 Range first address .. maxmem.
2121 - index is used as a byte counter.Range 0..1.
2122 - k is used to check the number of the characters
2123 that are typed by the user as new data.Range
2124 0..3.
2125 - first is the starting address.Range 0..maxmem.
2126 - times denotes how many words will be displayed.
2127 Range 0..maxmem.
2128 - done denotes that the operation must terminate.
2129 Range I or F.
2130
2131 The procedure calls :The nested procedure displaymix and the proce-
2132 dures decrhex , firstsecond.
2133 The procedure is called by : MONITOR.
2134
2135 *****
2136 *****
2137 *****
2138 *****
2139 *****
2140 *****
2141 *****
2142 *****
2143 *****
2144 *****
2145 *****
2146 *****
2147 *****

```

2161  
2162  
2163  
2164  
2165  
2166  
2167  
2168  
2169  
2170  
2171  
2172  
2173  
2174  
2175  
2176  
2177  
2178  
2179  
2180  
2181  
2182  
2183  
2184  
2185  
2186  
2187  
2188  
2189  
2190  
2191  
2192  
2193  
2194  
2195  
2196  
2197  
2198  
2199  
2200

procedure DISPLAY(var correct: boolean);

var  
max, index, k, first, times: integer;  
done: boolean;  
ch : char;

```

2225 (*****
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267

PROCEDURE displaymix.

FUNCTION
-----the procedure is used to print the contents of the desired
memory locations both in hex notation and as ASCII characters.

PARAMETERS
----- : None.

GLOBAL VARIABLES
----- : - first is local to the procedure DISPLAY.

LOCAL VARIABLES
----- : - n is the decimal value of the second half hex
character of the ASCII character.Range 0..15.
- num is the first half hex character of the ASCII
character and also it is used to calculate the
final decimal value to the character.Range 0..127.
- j is used as an index for the display of a group
of 8 words .Range 0..7.
- k is used as an index to display 16 characters
corresponding to 8 memory locations.Range 0..15.

the procedure calls : the procedures dechex, valuechar.
the procedure is called by : DISPLAY.

*****
procedure displaymix;

```





2344 end; (\*1 displaymix\*)

```
2345
2346
2347
2348
2349
2350 begin (*1*)
2351   done := false;
2352   firstsecond(first,times,correct);
2353   if odd(first) and correct then begin (*2*)
2354     er; (* checks if the first address is odd *)
2355     correct := false
2356   end (*2*);
2357   if correct then begin (*3*)
2358     skipblanks(ch);
2359     if ch <> 'w' then
2360       readln; (* checks if exists the parameter 'w' if not *)
2361       (* clears the input buffer *)
2362       max := first + 2 * times; (* calculates the final address *)
2363       if max > maxmem then
2364         max := maxmem; (* prevents to increase the subscript over *)
2365         (* the maximum memory *)
2366       if ch = 'w' then
2367         displaymix (* calls the displaymix to prints *)
2368         (* combined the contents of the memory in hex form and in
2369           equivalent ASCII characters *)
2370       else begin (*4*)
2371         repeat
2372           write(' '); (* left margin *)
2373           dechex(first,false); (* prints the hex value of the address *)
2374           write(' ', memory[first], memory[first + 1], ' < ');
2375           (* prints the contents of the memory *)
2376           if eoln(input) then begin (*5*)
2377             writeln;
2378             (* if eoln input denotes no input from the user so no
2379               change in the contents of the memory *)
```

```

2401 first := first + 2
2402 end else begin (*5, 6*)
2403 skipblanks(ch);
2404 if (ch = 'u') or eoln(input) then
2405   done := true (* The u character denotes that the user *)
2406   (* does not need any more the feature *)
2407 else begin (*7*)
2408   index := 0; (* byte index *)
2409   k := 0; (* counter to check the number of characters *)
2410   (* that the user will type for the new memory contents *)
2411   repeat
2412     if ch in validch then (* checks if the character is *)
2413       (* hex one *)
2414       memory[first+index] := ch
2415     else
2416       correct := false;
2417       index := (index + 1) mod 2;
2418       if index = 0 then
2419         first := first + 1; (* each memory address consists *)
2420         (* of two bytes *)
2421       k := k + 1;
2422       if k <= 3 then
2423         read(ch);
2424         until not correct or (eoln(input) and (k <> 3))
2425         or (k > 3);
2426         if odd(first) then (* checks if the user has typed less *)
2427           (* than the 4 characters as new data for the memory *)
2428             first := first + 1;
2429         end (* 7 *)
2430       end; (*6*)
2431       if first = max then (* checks for termination condition *)
2432         done := true;
2433       if not done and eoln(input)
2434         then readln;
2435       until done or (first >= max) or not correct

```

end (\* 4 \*)  
end (\* 3 \*)  
end; (\* 1 DISPLAY \*)

2465  
2466  
2467  
2468  
2469  
2470  
2471  
2472  
2473  
2474  
2475  
2476  
2477  
2478  
2479  
2480  
2481  
2482  
2483  
2484  
2485  
2486  
2487  
2488  
2489  
2490  
2491  
2492  
2493  
2494  
2495  
2496  
2497  
2498  
2499  
2500  
2501  
2502  
2503  
2504  
2505  
2506  
2507



|      |
|------|
| 2585 |
| 2586 |
| 2587 |
| 2588 |
| 2589 |
| 2590 |
| 2591 |
| 2592 |
| 2593 |
| 2594 |
| 2595 |
| 2596 |
| 2597 |
| 2598 |
| 2599 |
| 2600 |
| 2601 |
| 2602 |
| 2603 |
| 2604 |
| 2605 |
| 2606 |
| 2607 |
| 2608 |
| 2609 |
| 2610 |
| 2611 |
| 2612 |
| 2613 |
| 2614 |
| 2615 |
| 2616 |
| 2617 |
| 2618 |
| 2619 |
| 2620 |
| 2621 |
| 2622 |
| 2623 |
| 2624 |
| 2625 |
| 2626 |
| 2627 |

|      |
|------|
| 2585 |
| 2586 |
| 2587 |
| 2588 |
| 2589 |
| 2590 |
| 2591 |
| 2592 |
| 2593 |
| 2594 |
| 2595 |
| 2596 |
| 2597 |
| 2598 |
| 2599 |
| 2600 |
| 2601 |
| 2602 |
| 2603 |
| 2604 |
| 2605 |
| 2606 |
| 2607 |
| 2608 |
| 2609 |
| 2610 |
| 2611 |
| 2612 |
| 2613 |
| 2614 |
| 2615 |
| 2616 |
| 2617 |
| 2618 |
| 2619 |
| 2620 |
| 2621 |
| 2622 |
| 2623 |
| 2624 |
| 2625 |
| 2626 |
| 2627 |



```

2641
2642
2643
2644
2645     end; (*3*)
2646     if odd(first) or odd(second) then begin (*4*)
2647         er; (* prints an error message *)
2648         correct := true (* in order to prevent the MONITOR to *)
2649             (* print also another error message *)
2650     end (*4*)
2651     end; (* 1 FILE *)
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681

```

```

2705 (*****
2706
2707
2708
2709
2710 FUNCTION
2711 -----
2712         ex of command : < J 0CA0 <cr>
2713         The jump command is used to branch unconditionally to given
2714         even address.the procedure updates the counter register and
2715         the decimal equivalent ic also it checks for even address
2716         and if the address is in the range of the available memory.
2717         The procedure sets the variable quit to true denoting that
2718         the monitor program can be quitted and the variable alldone
2719         to false denoting that the user program can start the execu-
2720         tion.
2721
2722 PARAMIFIERS
2723 ----- by reference - correct denoting that the command was valid.
2724                                     Range T or F.
2725         - quit denoting that the monitor program can
2726           be quitted.Range T or F.
2727
2728 GLOBAL VARIABLES
2729 ----- : - ic is the decimal counter.Range 0 ..maxmem.
2730
2731 LOCAL VARIABLES
2732 ----- : - The parameters.
2733         - address is the unconditional address .Range 0...-
2734           maxmem.
2735
2736 the procedure calls : the procedures readress, setreq.
2737 the procedure is called by : MONITOR.
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747

```



```

2825 (*****
2826
2827 PROCEDURE MOVE.
2828
2829 FUNCTION
2830 ----- ex of command : < M <address1> <address2> < n >
2831
2832 The move command is used to move the contents of a block of
2833 memory from the source address 1 to the destination address 2.
2834 The n denotes the number of the bytes that will be moved.
2835 The procedure reads the two addresses and checks if the addres-
2836 ses are in the range of the memory .Also during the execution
2837 of the move command it checks for overflow in the memory
2838 due to large value of n.
2839
2840 PARAMETERS
2841 ----- : by reference - correct denotes that the command is
2842 correct .Range T or F.
2843
2844 GLOBAL VARIABLES
2845 ----- : - memory is the used real memory.
2846
2847 LOCAL VARIABLE
2848 -----
2849
2850 : - the parameter.
2851 - first is the first address(source) .Range
2852 0..maxmem.
2853 - second is the second address (destination ).
2854 Range 0..maxmem.
2855 - number is the parameter 'n ' of the command de-
2856 noting the number of bytes to be moved.Range
2857 0..maxmem.
2858 - j is used as the offset in the two base addresses
2859 during the execution.Range 0..number.
2860
2861 The procedure calls : The procedures firstsecond,readress.
2862 The procedure is called by : MONITOR.
2863
2864
2865
2866
2867

```

```

2881
2882
2883
2884
2885 *****
2886 *****
2887 *****
2888 *****
2889 *****
2890 *****
2891 *****
2892 *****
2893 *****
2894 *****
2895 *****
2896 *****
2897 *****
2898 *****
2899 *****
2900 *****
2901 *****
2902 *****
2903 *****
2904 *****
2905 *****
2906 *****
2907 *****
2908 *****
2909 *****
2910 *****
2911 *****
2912 *****
2913 *****
2914 *****
2915 *****
2916 *****
2917 *****
2918 *****
2919 *****
2920 *****
2921 *****
2922 *****

procedure MOVE(var correct: boolean);

var
j, first, second, number: integer;

begin (*1*)
firstsecond (first,second,correct);
if correct then (* correct and the two addresses *)
readaddress(number, correct);(*number of bytes for transfer*)
j := 0;
while correct and (j <= number) do begin (*2*)
if (first + j <= maxmem) and (second + j <= maxmem) then
memory[second + j] := memory[first + j];
j := j + 1;
end (*2*)
end; (* 1 MOVE *)

```

```

2945 (*****
2946
2947
2948 PROCEDURE NEXT.
2949
2950 FUNCTION
2951 ----- ex of command : < N [<n>]
2952 the NEXT command will cause the execution of the next 'n'
2953 machine instructions , starting at the current PC, and
2954 will display all registers ,current opcode for execution
2955 and the decimal value of the PC after each instruction
2956 execution.< n > is the range l..maxmem .If n is not given
2957 then 1 is assumed.
2958
2959 PARAMETERS
2960 ----- : by reference - correct denotes that the command is correct
2961 Range 1 or F.
2962 - quit denoting that the MONITOR program can
2963 be quited.Range 1 or F.
2964
2965 GLOBAL VARIABLES
2966 ----- : - next denotes the number of the instructions to
2967 be traced.Range 0..maxmem.
2968 - all done denoting that the user program can start
2969 the execution.Range 1 or F.
2970 - TRACE denoting that the trace mode is active from
2971 the next instruction.Range 1 or F.
2972
2973 LOCAL VARIABLES
2974 ----- : - times is the number of the instructions that
2975 will be traced .Range 1..maxmem.
2976
2977 The procedure calls : the procedure readaddress.
2978 The procedure is called by : MONITOR.
2979
2980 *****
2981 *****
2982 *****
2983 *****
2984 *****
2985 *****
2986 *****
2987 *****

```



```

3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040

    procedure NEXT(var correct, quit: boolean);

    var
        times: integer;

    begin (*1*)
        readaddress(times, correct);
        if correct then
            next := times
        else
            next := 1; (*default value is one *)
            quit := true; (* sets the quit flag in order to quit the *)
            alldone := false; (* resets the alldone flag in order the user's *)
            TRACE := true; (* the trace option is true *)
            correct := true
        end; (* 1 NEXT *)

```

# PROCEDURE REG

## FUNCTION

----- ex of command < R [< register name >]  
 The REGISFR command is used to examine or modify the  
 specified register.  
 The following registers names may be used in the command:  
 1. any of the sixteen 16 bit registers named R0,R1,R2..R15.  
 2 any of the sixteen 8 bit register named RH0,RL0,RH1,RL1,  
 ..RH7, RL7.  
 3 any of the eight 32 bit registers named RK0,RR2..RK14.  
 4 program counter register named RPC.  
 5 flag and control register named RFC.  
 If not register name is given ,all the registers R0..R15  
 PC and FCW will be displayed.If a register name is given  
 the specified register name will be displayed followed to  
 by its contents,followed by a space.If it is desired to  
 change the contents of that register , the new contents  
 be entered .A <cr> either alone or after the new contents  
 will cause the next register to be displayed.A 'Q' followed  
 by a <cr> will terminate the command.

## PARAMETERS

----- : by reference - correct denotes that the command is cor-  
 rect. Range 1 or F.

## GLOBAL VARIABLES

----- : - None.

## LOCAL VARIABLES

----- : - low is used to distinguish between low  
 byte registers and high byte registers

3121  
3122  
3123  
3124  
3125  
3126  
3127  
3128  
3129  
3130  
3131  
3132  
3133  
3134  
3135  
3136  
3137  
3138  
3139  
3140  
3141  
3142  
3143  
3144  
3145  
3146  
3147  
3148  
3149  
3150  
3151  
3152  
3153  
3154  
3155  
3156  
3157  
3158  
3159  
3160  
3161  
3162

- Range 0 or 8.  
- j is used in the for loop that prints the hex values of double words registers and also it is used in the loop that reads characters from the screen as new contents of the register. Range 0..3
- index is the final value of the register after the subtraction of the number d in the case of low byte register. Also it is used as index when the values of the registers are printed sequential.  
Range T or F.
- length denotes the type of the register byte, one word or double word. Range 0, 1 or 2.
- max denotes the number of characters that are permitted to be typed by the user as new contents for the register. Range 2 or 4 or 8.
- number is the number of the register.  
Range 0..15.
- val is the value of the contents of the register. Range 0..maxintery.  
In the case of double word registers the value of each one word register is retrieved and the are printed in concatenation.
- ch is used to read characters from the screen.
- name is a packed array used to save the name of the register during the scan and then to print together the value of the register and his name.
- byte denote that the register is byte type.  
Range T or F.

- done denotes that the procedure must return to the MONITOR program.Range I or F.
- change denotes that the user does not wish to make changes to the contents of the register.Range I or F.

The procedure executes the following in order :

- scans for character R after the first R of the command.If R is not encountered then it prints all the registers with their values and names and returns to the MONITOR.
- If R is encountered then checks for the second character if it is digit or not.If yes then scans if exists and second digit and calculates the final number of the register.
- If the second character is not digit then there are the following subcases :
  - second character L or H (RL ,RH)
  - second character R (RR)
  - second character F or P (RPC or RFC )
- scans for one or two digits after the second character and it prints the value of the register with the name.
- it prints a prompt 'x' and scans for input from the user .If not input is encountered then prints the value of the next register for all the cases except the for the registers RPC and RFC.
- It terminates if the range of the registers is covered or the user types 'Q'.
- If the user types new data for the register then updates his value.
- also it checks if the user has changed mode of operation and then it changes the stacks.

The procedure calls : the procedures skipblanks, REGISTER, retval-  
rea, dechex, setrea,STACKINIFCHANGE.  
the procedure is called by : MONITOR.



```

3305 if ch <> ' ' then
3306   number := number * 10 + ord(ch) - 48; (* two digits *)
3307   if number <= 15 then begin (*5*)
3308     index := number; (* index is valid *)
3309     length := 1; (* one word register *)
3310     max := 4
3311   end else (* four character register *) (*5*)
3312     correct := false
3313   end else if ch in ['H', 'L'] then begin (*3*) (*6*)
3314     name[] := ch;
3315     if ch = 'L' then begin (*6a*) (* a byte register RL or RH *)
3316       number := 0; (* Low byte register *)
3317       low := 8
3318     end; (*8a*)
3319     skipblanks(ch);
3320     if ch in ['0'..'9'] then begin (*7*)
3321       number := number + ord(ch) - 48;
3322       index := number;
3323       length := 0;
3324       max := 2
3325     end else (*7a*)
3326       correct := false (* a invalid character *)
3327     end else if ch = 'R' then begin (*8*) (*6*)
3328       name[] := ch;
3329       skipblanks(ch); (* a two word register *)
3330       if ch in ['0'..'9'] then begin (*9*)
3331         number := ord(ch) - 48;
3332         ch := ' ';
3333         skipblanks(ch);
3334         if ch <> ' ' then (* a second digit *)
3335           number := number * 10 + ord(ch) - 48;
3336           if (number <= 14) and not odd(number) then begin (*10*)
3337             index := number;
3338             length := 2; (* two word register *)
3339             max := 8
3340           end
3341         end
3342       end
3343     end
3344   end
3345 end
3346
3347

```



```

3361
3362
3363
3364
3365
3366
3367
3368
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3379
3380
3381
3382
3383
3384
3385
3386
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
3400
3402

    end else (* 8 character register *)(*10*)
    correct := false
    end (*9*)
    end else if ch in ['P', 'F'] then begin (*11*)(*8*)
    name[1] := ch;
    name[2] := 'C';
    if ch = 'P' then (* counter or FCW register *)
    number := 16
    else
    number := 19;
    index := number;
    length := 1;
    max := 4;
    skipblanks(ch);
    correct := true
    end; (*11*)
    if correct then begin (*20*)
    readln;
    repeat
    write(' ', name, index - low: 5, ' ');
    if length < 2 then begin (*21*)
    retvalrej(length, index, val);
    if length = 0 then
    byte := true;
    dechex(trunc(val),byte);(* prints the value of the *)
    (* register *)
    end else (*21*)
    for j := 0 to 1 do begin (*23*)
    retvalreg(1, index + j, val);
    dechex(trunc(val),byte)
    (* prints successively the values of one word registers *)
    end; (* 23 *)
    j := 0;
    val := 0;
    write(' < ');

```

```

3425 if eoln(input) then begin
3426   change := false; (* no change from the user is required *)
3427   writeln
3428 end else begin (*24*)
3429   while correct and not eoln(input) and (j < max) do
3430     begin (* 25 *)
3431       read(ch);
3432       if ch in ['0'..'9'] then (* digit *)
3433         val := val * 16 + ord(ch) - 48
3434       else if ch in ['A'..'F'] then
3435         val := val * 16 + ord(ch) - 55
3436       else begin (*26*)
3437         j := max + 1; (* invalid character *)
3438         done := true;
3439         if not ((ch = '0') and eoln(input)) then
3440           writeln('  ** No correct characters **')
3441         end; (*26a*)
3442         j := j + 1
3443       end; (*25*)
3444       writeln;
3445       change := true
3446     end; (*24*)
3447   if change and correct then begin (* 30 *)
3448     setren(length, val, index); (* set the register to the *)
3449     (* new value *)
3450     if ( r[fcw] [sn] and not system ) or
3451       ( not r[fcw] [sn] and system ) then
3452       STACKINTERCHANGE;
3453   if (index = 16) and (val < maxmem - 4) then
3454     ic := trunc(val); (* checks if the value of the *)
3455     (* register R 16 is in the bounds of the memory *)
3456   end; (* 30 *)
3457   if length = 2 then
3458     index := index + 2
3459   else

```

```

3481
3482
3483
3484
3485      index := index + 1;
3486      if ((length = 2) or (length = 1)) and (index >= 16)
3487      or (length = 0) and ( index -low >= 8) then
3488        done := true;
3489        if not done then
3490          readln
3491          until done or not correct
3492          end (* 20 if correct *)
3493        end; (* 2 correct command *)
3494        end; (* 1 REG *)
3495
3496
3497
3498
3499
3500
3501
3502
3503
3504
3505
3506
3507
3508
3509
3510
3511
3512
3513
3514
3515
3516
3517
3518
3519
3520
3521
3522

```

(\*\*\*\*\*  
3545  
3546  
3547  
3548  
3549  
3550  
3551  
3552  
3553  
3554  
3555  
3556  
3557  
3558  
3559  
3560  
3561  
3562  
3563  
3564  
3565  
3566  
3567  
3568  
3569  
3570  
3571  
3572  
3573  
3574  
3575  
3576  
3577  
3578  
3579  
3580  
3581  
3582  
3583  
3584  
3585  
3586  
3587

PROCEDURE SEND.

FUNCTION

----- The procedure transfers the image of the memory to an  
output file.

PARAMETERS

----- : None.

GLOBAL VARIABLES

----- : None.

LOCAL VARIABLES

- : - outouttext is the used as output file.
- adr1 is the initial address and also is used as memory index.Range 0 .. maxmem.
- k is used as byte counter.Range 0..1.
- adr2 is the final address for transfer .Range 0..maxmem.
- correct denotes if the address is correct. Range 1 or 0.

the procedure calls : The procedure firstsecond.  
The procedure is called by : MONITOR.

\*\*\*\*\*  
3573  
3574  
3575  
3576  
3577  
3578  
3579  
3580  
3581  
3582  
3583  
3584  
3585  
3586  
3587

{procedure SEND;

var

```

3601
3602
3603
3604
3605      outputtext: text;
3606      adr1 , adr2 , k : integer;
3607      correct : boolean;
3608
3609
3610      begin (*1*)
3611      rewrite(outputtext, 'outputfile');
3612      firstsecond (adr1, adr2, correct);
3613      if correct then (* correct address *)
3614      while adr1 <= adr2 do begin (*2*)
3615          for k := 0 to 1 do
3616              write(outputtext, memory[adr1][k]);
3617              adr1 := adr1 + 1;
3618          if adr1 mod 2 = 0 then
3619              write(outputtext, ' '); (* prints a space between the words *)
3620          if adr1 mod 16 = 0 then
3621              writeln ( outputtext) (* every 16 words feeds a line *)
3622          end; (*2*)
3623      reset(outputtext, 'outputfile');
3624      end; (*1 SEND *)
3625
3626
3627
3628
3629
3630
3631
3632
3633
3634
3635
3636
3637
3638
3639
3640
3641

```

```

3665 (*****
3666
3667
3668
3669
3670
3671
3672
3673
3674
3675
3676
3677
3678
3679
3680
3681
3682
3683
3684
3685
3686
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
3700
3701
3702
3703
3704
3705
3706
3707

```

PROCEDURE MONITOR.

-----

FUNCTION

-----

The procedure MONITOR is called as the first procedure from the main program and it executes the following functions in a for ever loop :

- finds the first character of the command of the user and it checks if the character is correct.If not it prints a error message and prints a new prompt '<|'.
- if the command is correct then calls the appropriate procedure to search the others parameters of the command and to execute the command if the parameters are valid.
- In the case of 'G' command it checks the status of the S/N flag with the existed mode of operation and if it is not consistent calls the procedure STACKINTERCHANGE.
- In the case of the commands 'G','J','N' checks if the program has been loaded .If it is not it prints a warning message.
- The user in order to terminate the program must type the character 'Q'.
- The procedure returns to the main program with the commands 'G', 'J', 'N' and 'Q'.

PARAMETERS

----- : None.

GLOBAL VARIABLES

-----

- all done denotes the end of the user's program. Range T or F.
- finished denotes that the main program must terminate.Range T or F.



```

3721
3722
3723
3724
3725
3726
3727
3728
3729
3730 LOCAL VARIABLES
3731 -----: - ch is used to read characters from the screen.
3732           Range ASCII characters.
3733           - correct denotes that the command is correct.
3734             Range T or F.
3735           - quit denotes that the procedure must return to
3736             the main program.
3737
3738 The procedure calls : The procedures BREAK, COMPARE, DISPLAY, FILL,
3739 JUMP, load, MOVE, REG, SEND.
3740 The procedure is called by : the main program.
3741
3742
3743
3744 *****
3745 *****
3746 *****
3747 *****
3748 *****
3749 *****
3750 *****
3751 *****
3752 *****
3753 *****
3754 *****
3755 *****
3756 *****
3757 *****
3758 *****
3759 *****
3760 *****
3761 *****
3762 *****

```

```

procedure MONITOR;
var
ch: char;
correct, quit: boolean;

begin (*1*)
quit := false;
correct := true;
write(' ');
while not quit do begin (*2*)
    ch := ' ';

```

```

3786 skipotanks(ch);
3787 if ch in ['R', 'C', 'D', 'F', 'G', 'J', 'M', 'Q', 'R',
3788         'N', 'L', 'S'] then begin (*3*)
3789     if (ch in ['J', 'G', 'N']) and not loprogram then
3790         writeln(' ** Program is not loaded **')
3791     else
3792         case ch of
3793             'R':
3794                 BREAK(correct);
3795             'C':
3796                 COMPARE(correct);
3797             'D':
3798                 DISPLAY (correct);
3799             'F':
3800                 FILL(correct);
3801             'G':
3802                 begin (*3*)
3803                     alldone := false; (* permits the simulator to start *)
3804                     quit := true;
3805                     correct := true;
3806                     end; (*3*)
3807             'J':
3808                 JUMP(correct, quit);
3809             'L':
3810                 load;
3811             'M':
3812                 MOVF(correct);
3813             'N':
3814                 NXT(correct, quit);
3815             'Q':
3816                 begin (*4*)
3817                     quit := true;
3818                     alldone := true;
3819                     finished := true
3820                     end; (*4*)
3821
3822
3823
3824
3825
3826
3827

```

```

3841
3842
3843
3844
3845
3846
3847
3848
3849
3850
3851
3852
3853
3854
3855
3856
3857
3858
3859
3860
3861
3862
3863
3864
3865
3866
3867
3868
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3880
3881
3882

```

```

'R':
    KFG(correct);
'S':
    {SEND} null
    end (* case *)
end else if ch <> ' ' then (*3*)
    correct := false;
    if (eoln(inout) or not correct) and not quit then
        readln; (* skips a line clearing the input buffer *)
    if not correct then begin (*4*)
        writeln(' ** Invalid command **');
        write(' < ');
    end else if not quit then (*4*)
        write(' < ');
    correct := true (* sets again the flag correct to true for *)
    (* the next command *)
end; (*2*) (* while loop *)
end; (* 1 MONITOR *)

```

```

3905 (*****
3906
3907         MAIN PROGRAM
3908
3909 The main program :
3910 - sets the maximum permitted number of statements to be executed.
3911 - enters a while for ever loop calling either the MONITOR or
3912   the EXECUTE procedure which is the main procedure for the si-
3913   mulator.
3914 - If the user types the command Q in the MONITOR the program stops.
3915
3916 *****
3917
3918
3919
3920
3921 (*main program*)
3922 begin (*1*)
3923   stlimit(3000000);
3924   boot;
3925   while not finished do begin (*2*)
3926     MONITOR;
3927     if not finished then
3928       EXECUTE
3929     end (*2*)
3930   end. (* 1 main program *)
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947

```

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17 (*****
18 *
19 *
20 *
21 *
22 *
23 *
24 *
25 *****
26
27
28 program LUAD(input, output);
29
30
31
32 const
33     maxmem = 1800; (* available virtual memory in the simulator *)
34     maxlen = 15; (* maximum length of the user's program filename *)
35     mar = ' '; (* left margin for output messages *)
36
37
38 type
39     echo = packed array [1..maxfilen] of char;
40
41
42

```

66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108



```

121
122
123
124
125 var
126     code, ready: text;
127     (* are the used files for the input code and output object file *)
128     codefile, readyfile: echo; (* names of the aboves files *)
129     alldone: boolean;
130     (* if error condition occurs terminates the execution *)
131     valid, small: restricted;
132     (* set of characters to check
133     if the input stream of characters is valid *)
134     quit : boolean ; (* denotes that the user wishes to quit
135                        the program *)
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162

```

```

186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228

PROCEDURE error.

FUNCTION
----- Prints an error message depending on the value of the
parameter 'n'.

PARAMETERS
----- : by value - 'n' denotes the type of error..
Range 1..2.

GLOBAL VARIABLES
----- : None.

LOCAL VARIABLES
----- : None.

The procedure is called by : skipcomments.

*****

procedure error(n: integer);
begin (*1*)
case n of
1:
    writeln(mar, 'Invalid character encountered in the input file ');
2:
    writeln(mar, 'Size of code greater than maximum memory ');
end (*case*)
end; (*1 error *)

```

```

301
302
303
304
305 (*****
306
307
308
309
310

```

# PROCEDURE skipcomments.

## FUNCTION

```

----- The procedure performs the following in order:
- prints a message to the terminal.
- scans for the character 'Q' and < cr > or the input
  filename.
- if the user types the character 'Q' then the procedure
  returns to the main program and terminates the main
  program.
- if the user types the filename then opens the file and
  creates an object file skipping the comments and the
  spaces from the input hex file.
- it checks for small characters changing them to capital.
- it checks for valid hex characters ['0'..'9','A'..'F'].
- it checks for the size of the input code.

```

## PARAMETERS

```

----- : by reference - quit denotes that the user has typed
the character 'Q'.Range I or F.
- alldone denotes that an error condition
has occurred.Range I or F.

```

## GLOBAL VARIABLES

```

----- : - ready is the output file.
- readyfile is the output filename.
- code is the input file.
- codefile is the input filename.

```

## LOCAL VARIABLES

```

----- : - ch is used to read characters from the terminal,

```

```

file.Range ASCII characters.
- zero is used to write a zero character to the out-
  put file in the case that the byte count of the
  input file is odd.
- ic is used as memory index counting the size of
  the input file.Range 0..maxmem.
- count is used to count the characters of the input
  filename and also it is used as a byte counter.Range
  0..15.

```

```

376 The procedure calls : the procedure error.
377 The procedure is called by : the main program.
378

```

```

379 *****
380

```

```

381 procedure skipcomments (var quit , alldone : boolean );
382

```

```

383 var
384   ch, zero: char;
385   ic, count : integer ;
386

```

```

387
388
389
390
391   begin (**)
392     count := 0;
393     ic := 0;
394     zero := '0'; (*'0'*)
395     readyfile := 'executecode'; (* object filename *)
396     (*output file of this program
397     and input file for the simulator program*)
398     writeln;
399     writeln('*****');
400

```

```

421 writeln(mar, '1. welcome to Z8000 simulator ');
422 writeln(mar, '2. Enter the filename of code file');
423 writeln(mar, '3. Press return and wait');
424 writeln(mar, '4. Type 'ex' and return to start the simulator');
425 writeln(mar, '5. To quit the present program type 'Q' followed ');
426 writeln(mar, ' by return. ');
427 writeln('*****');
428
429 read(ch);
430 if (ch = 'Q') and (count = 0) then
431   quit := true; (* means to quit the program *)
432 if not quit then begin (* 2 *)
433   count := count + 1;
434   codefilelcount := ch;
435   end; (* 2 *)
436 until eoln(input) or (count = maxfilen) or quit;
437 (* it reads from the terminal the input filename till
438 eoln occurs or the length of the filename equals the
439 maximum permitted *)
440 if not quit then begin (* 3 *)
441   reset(code, codefile); (* ready the codefile for reading *)
442   rewrite(ready, readyfile); (* creates the output file *)
443   count := 0;
444   repeat
445     if not eof(code) then
446       read(code, ch);
447       if (ch <> ' ') and (ch <> ';') then begin (* 4 *)
448         if ch in small then
449           ch := chr(ord(ch) - 32); (*no capital character*)
450           (* and changes the character to capital one *)
451         if ch in valid then
452           begin(* 5 *)
453             write(ready, ch) (* checks if the character is valid *)
454             (* hex one *)

```

```

486 else begin (* 6 *)
487   error(1);(*invalid character*)
488   alldone := true (* so terminates the procedure *)
489   end; (* 6 *)
490   count := (count + 1) mod 2; (* 0..1 bytes *)
491   if count = 0 then begin (* 7 *)
492     ic := ic + 1; (* counts the size of the input code
493       and checks if it is less than the
494       available memory *)
495   end (* 7 *)
496   end (* 4*);
497   if eoln(code) or (ch = ';') then (* skips comments *)
498     readln(code)
499   until (ic = maxmem + 1) or alldone or eof(code);
500   ic := ic - 1; (* decrement ic to the correct value *)
501   if not eof(code) and (ic = maxmem) and not alldone
502     then begin (* 8 *)
503     error(2);(* size of code greater than permitted *)
504     alldone := true
505     end; (* 8 *)
506   if count = 1 then (* checks if exists odd number of bytes *)
507     write(ready, zero);
508   end; (* 3 *)
509   end; (* 1 skipcomments *)

```





606 end. (\* J main program \*)

607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627

```

1
2
3
4
5
6
7
8
9
10
11 ;          PROGRAM TRAP
12
13 ; This program provides the hex code for the Program Status
14
15 ;Area and the code for the trap handlers subroutines. The
16
17 ;purpose of the code is to prove that the architecture can support
18
19 ;any trap handler subroutine. So in this program the policy is to
20
21 ;execute the priviledges opcodes that are called in the normal mode.
22
23
24 0000;          RESERVED
25 0000;
26 4000;          UNIMPLEMENTED INSTRUCTION
27 0718;
28 4000;          PRIVILEGED INSTRUCTION
29 071A;
30 4000;          SYSIFM CALL
31 0770;
32 4000;          AVAILABLE FOR OTHER CASES
33 0000;
34 4000;
35 0000;
36 7A00;          HALT INSTRUCTION FOR UNIMPLEMENTED INSTRUCTION
37 5C09 0A05 07A0; SAVES THE CONTENTS OF THE REGISTERS
38 ;             R10..R14 BECAUSE WILL BE USED BY THE
39 ;             TRAP INSTRUCTIONS.
40
41
42

```

```

66 0CE1 5A00; (MP MEMORY(R14) WITH 3A H
67 ; THE OPCODES STARTING WITH 3A AND 3B
68 ; ARE TWO WORDS LONG OPCODES
69 5E06 0736; IF ZERO JUMP TO 0736 H
70 0CF1 3B00; CMP MEMORY(R14) WITH 3B H
71 5E06 0736; IF ZERO JUMP TO 0736H
72 5E08 0758; ELSE JUMP TO 0758H
73 A9E3; INCRFASE R14 ← R14 + 4
74 21F0; R13 ← MEMORY(R14) SO R13 HAS THE VALUE OF THE PC
75 21DC; R12 ← MEMORY(R13) SO FIRST WORD OF THE OPCODE
76 A9D1; R13 ← R13 + 2
77 21DB; R11 ← MEM (R13) SO THE SECOND WORD OF THE OPCODE
78 A9D1; R13 ← R13 + 2 SO PC = PC + 2
79 2FED; STORE NEW PC IN THE MEMORY(R14)
80 6F0C 0752; STORE THE OPCODE TO THE NEXT BLANK PLACES
81 6F0B 0754; STORE THE SECOND WORD OF THE OPCODE TO THE
82 ; NEXT BLANK SPACE
83 5C01 0A05 07A0; RESTORE THE REGISTERS R10..R14.
84 0000
85 0000
86 7B00; IREF
87 ; THIS NEXT PORTION OF THE OPCODE IS USED IN THE CASE
88 ; THAT THE PRIVILEGED INSTRUCTION IS ONE WORD OPCODE
89 ; AND SO THE TRAP CODE INCREASES THE IC ONLY BY TWO
90 ; MEMORY LOCATIONS AND COPIES ONLY ONE WORD OPCODE
91 A9F3; R14 ← R14 + 4
92 21FD; R13 ← MEMORY(R14)
93 21DC; R12 ← MEMORY(R13)
94 A9D1; PC ← PC + 2
95 2FED; STORE NEW PC IN THE STACK AREA
96 6F0C 076C; COPY THE OPCODE TO THE BLANK SPACE
97 5C01 0A05 07A0; RESTORE THE REGISTERS R10..R14.
98 0000
99 7B00; IREF
100
101
102
103
104
105
106
107
108

```

```

121
122
123
124
125 AIFE;          FOR THE CASE OF SYSTEM CALL
126 ;             LD R14 ← R15
127 A9E3;          R14 ← R14 + 4
128 21ED;          R13 ← MEM (R14) SO HAS THE VALUE OF PC
129 A9D2;          R14 ← R14 + 3
130 2FFD;          STORE THE NEW PC IN THE STACK AREA
131 5C01 0A05 07A0; R13 ← R14 + 3
132 7B00;          RESTORE THE REGISTERS R10..R14.
133             TRFI
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153

```

```

6 *
7 *
8 *
9 *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 *
25 *
26 *
27 *
28 *
29 *
30 *
31 *
32 *
33 *
34 *
35 *
36 *
37 *
38 *
39 *
40 *
41 *
42 *
43 *
44 *
45 *
46 *
47 *
48 *

PROGRAM SIMULATOR.P

This is the virtual version and they are listed only
the new procedures and the changes to the old ones

(*****
(*****
(*****

CONSTANT DECLARATION

(*****
(*****
(*****

const
maxvirtual = 32000 ; (* denotes the maximum virtual memory *)
maxmem = 2600;(* available real memory *)
nsaarea = 25600;(* the starting address of the PSA area *)
systemstackpointer = 50000;(* default value for the system stack
pointer *)
range = 255; (* denotes the size of each segment of
the real memory *)

```



```

61
62
63
64
65 type
66     part = record
67         upper : integer; (* denotes the high address of the
68                             segment *)
69         low : integer; (* denotes the low address *)
70         reference : boolean; (* denotes if the memory contained
71                                 in the segment has been referenced *)
72         changed : boolean; (* denotes if the memory has been *)
73                             (* changed *)
74         load : boolean ; (* denotes if the segment has been loaded
75                             with code *)
76     end;
77     segment : array [0..9] of part; (* is the division of the memory
78   to ten segments three are used for the stackpointers and
79   the PS area and seven are used for the code and data of the
80   running program *)
81     last : integer; (* denotes the number of most recently loaded
82                       segment from the code *)

```

```

126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169

PROCEDURE setsegment

FUNCTION
----- the procedure is used to set the segments to the initial
          values and status. All the segments except the segment 0,
          7,8,9 are not loaded till a need is arised.
          the segment 0 is loaded with the first part of the program,
          the segment 7 is loaded with the Program Status Area and
          the trap handlers subroutines, segment 8 and 9 are loaded
          with the stack areas.

PARAMETERS
----- : by value - number denotes the number of the segment.
          Range 0..9.
          - address denotes the high address of the seq-
            ment. Range 0..maxvirtual (32 or 64 k as will
            be specified by the constant maxvirtual )

GLOBAL VARIABLES
----- : segment denotes the used segment.

LOCAL VARIABLES
----- : - The parameters.

The procedure is called by : load,hont.
The procedure calls : None.

*****

procedure setsegment (number,address : integer);

```

```

181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223

begin(*1*)
  with segment [ number ] do begin (*2*)
    upper := address;
    low := address - range; (*sets the low bound of the
      the segment. The range constant has value 255.*)
    if low < 0 then begin (*3*)
      load := false; (* means that is initialization condition *)
      reference := false; (* no reference *)
      changed := false; (* no change *)
    end else begin (*3,4*)
      load := true; (* the segment is loaded *)
      reference := true;
    end; (*4*)
    changed := false; (* and in the two case the contents
      of the segment have not changed *)
  end; (*2*)
end; (*1 setsegment *)

```

# PROCEDURE update.

## FUNCTION

----- The procedure is called when a changed segment must be loaded with new data from the memory. Because the operation of update the file is costly the procedure also checks if the contents and of another segment have changed and if yes then transfers the contents of the segment to the memory changing his status to no changed.

## PARAMETERS

----- : by value - num denotes the number of the segment to be transferred to the memory. Range 0..9.

## GLOBAL VARIABLES

----- : segment is the used segment.

## LOCAL VARIABLES

----- :  
 - code, temp are the two used files to perform the operation, code is the original file and temp is the temporary file which is removed from the file system after the operation.  
 - k is used as index in for loop to check if other segments are also changed after the transferring of the first one.  
 - j is used as address counter during the transferring operation. Range 0..maxvirtual.  
 - address, adrl are used to denote the starting address for the transferring. Initial the procedure assumes that the initial address is the low bound of the transferred as parameter segment but it also checks if exists and another one and the low bound address of the new segment is assigned

```

301      to the variable adr1.
302      - first denotes the low bound of the virtual address
303      and depends on the number of the segment.Range
304      0..maxmem.
305      - numnew,number are used to denote the numbers
306      of the segments for transfer.Range 0..9.
307      - ch,chl are used to read and to write two cha-
308      racters so one memory unit.
309      - done denotes if the operation has been accom-
310      plished.Range T or F.
311
312
313
314
315

```

```

316 The procedure is called by : findandload.
317 The procedure calls : None.
318

```

```

319 *****
320

```

```

321
322
323 procedure update ( num : integer );

```

```

324
325 var
326   code,temp : text;
327   k,i,address : integer;
328   first,number,adr1 : integer;
329   ch,chl : char;
330   done : boolean;
331   numnew : integer;

```

```

332 begin (*1*)
333   j := 0; (* resets the address counter *)
334   adr1 := 1000000;
335   rewrite (temp,'tem');
336   reset (code, 'executecode');
337   address := segment [num].low;
338   (* assumes that the lowest address for trasferring is the
339

```

```

numnew := num;
repeat
  for number := 0 to 6 do begin (* 2 *)
    if segment[number].changed then
      adr1 := segment[number].low;
    (* checks if exists and another segment with lowest
       address *)
    if adr1 < address then
      begin (* 3 *)
        address := adr1;
        (* if it exists the changes the starting address to the
           low bound of the new segment *)
        numnew := number;
        adr1 := 1000000;
      end; (* 3 *)
    end; (* 2 *)
    segment[numnew].changed := false;
    (* changes the status of the segment *)
    while (j < address) and not (eof(code)) do begin (* 4 *)
      read(code, ch, chl);
      write(temp, ch, chl);
      (* transfers unchanged file to the temporary *)
      j := j + 1;
    end; (* 4 *)
    while (j < address) do begin (* 5 *)
      write (temp, '00');
      (* fills the file with zeros, means that this part of
         the memory has not changed and used *)
      j := j + 1;
    end; (* 5 *)
    first := num * (range + 1);
    (* calculates the virtual address *)
    for j := first to (segment[numnew].upper-segment[numnew].low
      + first) do

```



```

421 write(temp,memory[i]);
422 (* transfers the contents of the segment *)
423 j := segment[numnew].upper;
424 (* updates the address counter *)
425 address:= 1000000;
426 done := false;
427 for k := 0 to 6 do begin (* 6 *)
428   done := segment[k].changed or done;
429   (* checks if exist and another segment for transferring *)
430   if segment[k].changed then
431     if address > segment[k].low then begin (* 7 *)
432       address := segment[k].low;
433       numnew := k;
434     end; (* / *)
435   end; (* 6 *)
436   adr1:= 10000000;
437   until not done;
438   while not eof(code) do
439     begin (* 8 *)
440       read(code,ch,ch1);
441       (* transfers the rest of the file *)
442       write(temp,ch,ch1);
443     end;(* 8 *)
444     reset (temp,'tem');
445     rewrite(code,'executecode');
446     while (not eof(temp)) do begin (* 9 *)
447       (* starts the rotation of the updated file *)
448       read(temp,ch,ch1);
449       write(code,ch,ch1);
450     end;(* 9 *)
451     remove ('tem');
452     (* deletes the file from the file system *)
453     end;(* 1 update *)
454
455
456
457
458
459
460
461
462

```

## PROCEDURE Loadnew.

The procedure loads a specified address in a segment. The size of the segment is 255 bytes and there is possibility some part of the range from the initial address till the upper bound to have been loaded to another segment and during the loading it checks if this condition occurs and then terminates the operation.

- ind denotes the starting address. Range 0 .. maxvirtual.
- number denotes the number of the segment. Range 0..6.

## - segment denotes the used segment.

- ch,chl are used to read and write two characters from the file and to the memory so one memory unit.
- exist denotes if the address already exists loaded to another segment. Range 1 or F.
- j is used as address counter.Range 0 .. maxvirtual.
- k is used as address counter during the operation of the transferring the data from the memory to the segment.Range 0..255.
- l is used in the checking mechanism for the loading address if it is loaded to another

```

601
602
603
604
605      segment and is used as segment number.Range
606      0..0.
607      - code is the text file.
608
609      The procedure calls : None.
610      The procedure is called by : findandload.
611
612
613      *****)
614
615
616
617      procedure loadnew (ind,number : integer);
618
619      var
620          l,k,j : integer;
621          ch,ch1 : char;
622          found : boolean;
623          code : text;
624
625      begin (*1*)
626          j := 0;
627          (* j is used as address counter *)
628          found := false;
629          (* the variable found is used to check if part of the segment
630             that will be loaded is already loaded in some other
631             segment and so to terminate the loading procedure *)
632          reset (code,'executecode');
633          while (j < ind) and not eof (code) do
634              begin (*2*)
635                  read (code,ch,ch1);
636                  (* the procedure assumes that already the memory has
637                     been update if the used segment has been changed *)
638                  j := j + 1;
639                  (* it reads from the file till the desired address *)
640
641
642

```

```

606 k := 0;
607 while not eof(code) and (k <= range) and not found do
608   begin (* 3 *)
609     read(code,ch,ch1);
610     (* starts to read from the file and to load direct
611      to the segment *)
612     memory[number * (range + 1) + k] [0] := ch;
613     memory[number * (range + 1) + k + 1] [1] := ch1;
614     (* the offset address of the segment in the
615      real memory depends on his number * range + 1
616      which has the value of 255.*)
617     j := j + 1;
618     k := k + 1;
619     (* j is address counter and is used to load
620      the correct size of memory to the segment *)
621     found := false;
622     l := 0;
623     repeat
624       l := l + 1;
625       if l <> number then
626         found := (ind >= segment[l].low) and
627                   (ind <= segment[l].upper);
628       (* checks if the just loaded address is already
629        in some other segment because exist possibility
630        two segments to have overlap *)
631     until (l = 6) or found;
632     (* terminates the search if the address is found or all
633      the segments has been checked *)
634     if found then
635       j := j - l;
636     (* if the address was found then the j which is address
637      counter is decreased *)
638     end;(* 3 *)
639     (* if eof is encountered then the usedr has use the move

```

```

command to move part of the memory to other address
or has used load or store instruction to unloaded
memory *)
if eof(code) and ( j < ind ) then
  j := ind;
while ( k < range) and not found do
  begin (* 4 *)
    memory [number * (range + 1) + k] := '00';
    k := k + 1;
    j := j + 1;
    l := 0;
  repeat
    l := l + 1;
    if l <> number then
      found := (ind >= segment[l].low) and
        (ind <= segment[l].upper)
  until (l = 6) or found;
  if found then
    j := j - 1;
  (*if the file was smaller than the desired locations
  to be loaded to the memory then the memory is
  filled with zeros *)
  end; (* 4 *)
  with segment[number] do begin (* 5 *)
    low := ind;
    (* updates the parameters of the segment *)
    upper := j;
    changed := false;
    reference := true;
    load := true;
  end; (* 5 *)
end; (* 1 loadnew *)

```

```

786 (*****
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829

```

PROCEDURE findandload.

FUNCTION

----- The procedure searches the segments to find which one  
will be used to load the new desired contents of the  
memory. It starts the checking with the last loaded  
segment and if a not referenced or not loaded segment  
is encountered then if the segment has been changed  
calls the procedure update to update the memory and  
the procedure loadnew to load the new address.

PARAMETERS

----- : by value - ind is the desired initial address  
for loading. Range 0..maxvirtual.  
by reference - number denotes the used segment  
to be loaded. Range 0..b.

GLOBAL VARIABLES

----- : segment denotes the used segment.

LOCAL VARIABLES

----- : - i is used as segment number in the searching  
mechanism. Range 0..b.

The procedure is called by : check  
The procedure calls : loadnew, update.

\*\*\*\*\*)



```

841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882

```

```

procedure findandload ( ind : integer; var number : integer);
var
  j : integer;

begin (* 1 *)
  j := last ;
  (* starts the search from the last referenced segment *)
  repeat
    j := ( j + 1 ) mod 7;
    (* repeat the search in mod mode skipping the
       segments 0..9 that remain continuous loaded on the
       memory *)
  until ((not( segment[j].load )) or ( j = last ) or
        (not( segment[j].reference)));
  (* searches to find a segment not loaded or not referenced *)
  number := j;
  (* assigns the found value to the parameter number in
     order to return to the calling procedure *)
  if not segment[j].load then
    loadnew(ind,j)
  (* if the segment is not loaded then loads the desired
     address to the segment *)
  else begin (* 2 *)
    if segment[j].changed then
      update (j );
    (* if the segment has been changed with the fuction
       setmem then the contents of the segment must be
       transferred to the memory before it loads the new
       address *)
    loadnew (ind,j);
  end; (* 2 *)
  last := number; (* sets the last to his new value if
the segment number was different *)

```



906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948

```

961
962
963
964
965 (*****
966
967
968
969
970 FUNCTION
971 ----- The procedure checks if the desired address is loaded
972 in the some segment.If it is loaded then calculates
973 the virtual address and it also returns the initial
974 address to the calling procedure.If the address is not
975 loaded then calls the the procedure findandload to
976 load the address.
977
978 PARAMETERS
979 ----- : by reference - ind denotes the desired address.
980 Range 0..maxvirtual.
981 By the procedure is changed as
982 virtual.Range 0..maxmem.
983 - number is the used segment.Range
984 0 ..9.
985 - correction is the initial address.
986 Range 0..maxvirtual.
987
988 GLOBAL VARIABLES
989 ----- : - segment is the used segment.
990
991 LOCAL VARIABLES
992 ----- : - the parameters.
993 - exist denotes if the address exists.Range
994 I or F.
995 - indinitial is used to save the initial ad-
996 dress.Range 0..maxvirtual.
997
998 The procedure is called huv : map , setmem,COMPARE,FILL,MOVE,
999 DISPLAY,EXECUTE.
1000 The procedure calls : the procedure findandload.
1001
1002
1003

```

```

1026 *****
1027 *****
1028 *****
1029 *****
1030 *****
1031 *****
1032 *****
1033 *****
1034 *****
1035 *****
1036 *****
1037 *****
1038 *****
1039 *****
1040 *****
1041 *****
1042 *****
1043 *****
1044 *****
1045 *****
1046 *****
1047 *****
1048 *****
1049 *****
1050 *****
1051 *****
1052 *****
1053 *****
1054 *****
1055 *****
1056 *****
1057 *****
1058 *****
1059 *****
1060 *****
1061 *****
1062 *****
1063 *****
1064 *****
1065 *****
1066 *****
1067 *****
1068 *****

procedure check (var ind , correction , number : integer );

var
    exist : boolean;
    indinitial : integer;

begin (* 1 *)
    exist := false; (* assumes that the address does not exist*)
    indinitial := ind; (* assigns the initial real address
        to the variable indinitial in order
        to return as correction to the calling
        procedure *)
    number := last;(* starts the search from the last referenced
        segment *)
    repeat
        with segment [number] do begin (* 2 *)
            if (ind >= low ) and (ind <= upper ) and (low >= 0) then
                (* checks if the address is in the boundaries of the
                segment *)
                begin (* 3 *)
                    exist := true;
                    (* sets the variable exist in order to terminate
                    the search *)
                    reference := true;
                    end else (* 3 *)
                        reference := false;
                        (* if the address is not in the segment the resets
                        the reference flag *)
                end
            end
        end
    until exist;
end check;

```



```

procedure setmem(index: real; lbyte, length: integer;
               value : real );
var
  val: real;
  ind, byt, temp, j: integer;
  correction, number : integer;
  *
  A
  *
  check (ind, correction , number );
  memory(lind)lbytl := chr(temp);(*set the memory*)
  ind := correction;
  segment[number].changed := true;
end;(*l setmem*)

```

```

1201
1202
1203
1204
1205 procedure map(index: real; lhbyte, length: integer;
1206               var res : real);
1207 var
1208     j, k, ind, byt: integer;
1209     mul: real;
1210     correction, number : integer;
1211
1212     begin(*l*)
1213         *
1214         *
1215         *
1216
1217     for j := length downto 1 do begin(*3*)
1218         check (ind,correction, number );
1219         valuechar(memory[ind]fbyt,k);
1220         ind := correction;
1221     . end;(*l map *)

```

```

1260      *
1261      *
1262      *
1263
1264
1265      setreq(1, maxvirtual, nspoff);(*default value for the stackpointer*)
1266      system := false;(* normal is assumed as default mode of operation*)
1267      setreq(1, systemstackpointer, 23); (* default value for the system*)
1268
1269      ic := 0;(* sets the decimal counter to zero*)
1270      (* stack-pointer *)
1271      (* the user can change the counter if he changes the RPC *)
1272      setreq(1, psaarea, psaoft);(* sets the psaoft register to the address*)
1273      (* of the PSA area *)
1274
1275      reset(psacode, 'trapvirtual');
1276      byte := 0;
1277
1278      i := 1792; (* address of the PSA area *)
1279      while not eof(psacode) do begin (*2*)
1280          if not eoln(psacode) then begin(*3*)
1281              read(psacode, ch);
1282              memoryv[i] byte := ch;
1283              byte := (byte+1) mod 2;
1284              if byte = 0 then
1285                  i := i+1;
1286              end(*3*)
1287          else
1288              readln(osacode);
1289          end;(*2*)
1290
1291      for j := 0 to 6 do
1292          setsegment(j, 0);
1293          setsegment(9, 32000);
1294          setsegment(8, 30000);
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308

```



```

1321      setsegment (7, 25855);
1322      last := 0;
1323      end; (* 1 boot *)
1324      procedure load;
1325      setsegment (0, range);
1326      end; (* 1 load *)

```

```

1327      procedure EXECUTE;
1328      var
1329      correction, number : integer;

```

```

1330      begin (* 1 *)
1331      repeat
1332      check (ic, correction, number);
1333      decode(opcode, memory(ic));
1334      ic := correction;
1335      if TRACF then begin (* 2 *)
1336      *
1337      *
1338      *
1339      if (ic < 0) or (ic > maxvirtual - 3) then
1340      error(1); (* out of memory range *)
1341      end; (* 1 EXECUTE *)

```

```

1342      procedure readaddress(var n: integer; var correct: boolean);
1343      *
1344      *
1345      *

```

and (ch in '0'..'9'..'A'..'F') then

1386  
1387  
1388  
1389  
1390  
1391  
1392  
1393  
1394  
1395  
1396  
1397  
1398  
1399  
1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1420  
1421  
1422  
1423  
1424  
1425  
1426  
1427  
1428

```
correct := false;
(* checks if the final result is correct *)
end; (* l readaddress *)

procedure COMPARE(var correct: boolean);

var
j, first, second, times: integer;
adr1,adr2,cor1,cor2,number : integer;

begin (*l*)
  *
  *
  *
  and (j <= times) do begin (*3*)
    adr1 := first + j;
    adr2 := second + j;
    check (adr1,cor1,number);
    check (adr2,cor2,number);
    if memoryfadr1 <> memoryfadr2 then begin (*4*)
      write(' * first address ');
      dechex(first + j,false);(*write the first address*)
      write(' ',memoryfadr1, ' ');
      (* prints the contents of the memory *)
      write(' Second address ');
      dechex(second + j,false); (* write the second address *)
      writeln(' ',memoryfadr2);
      (* prints the contents of the second address*)
    end; (*4*)
```

```

1441     j := j + 1
1442   end (*3 while *)
1443   end; (* 1 COMPARE *)
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455

```

```

1456 procedure DISPLAY(var correct: boolean);
1457
1458
1459
1460

```

```

1461 var
1462   max,index, k, first, times: integer;
1463   done: boolean;
1464   ch : char;
1465   adr1,adr2,correction,number : integer;
1466

```

```

1467   begin (*1*)
1468     *
1469     *
1470     *
1471     k := 0;
1472     for j := 0 to 7 do begin (*2*)
1473       adr1 := first + k;
1474       adr2 := first + k + 1;
1475       check (adr1,correction,number);
1476       check (adr2,correction,number);
1477       write(' ', memory[adr1], memory[adr2]);
1478       k := k + 2 (* prints one word at a time *)
1479       (* in each loop it prints 8 words *)
1480     end; (*2*)
1481     write(' '); (* separator from the equivalent characters *)
1482     for j := 0 to 15 do begin (* 3 *)
1483       adr1 := first + j;
1484       check (adr1,correction,number);
1485       valuechar(memory[adr1][0],num);
1486
1487
1488
1489
1490
1491
1492
1493

```

1506  
1507  
1508  
1509  
1510  
1511  
1512  
1513  
1514  
1515  
1516  
1517  
1518  
1519  
1520  
1521  
1522  
1523  
1524  
1525  
1526  
1527  
1528  
1529  
1530  
1531  
1532  
1533  
1534  
1535  
1536  
1537  
1538  
1539  
1540  
1541  
1542  
1543  
1544  
1545  
1546  
1547  
1548

```

begin (*1*)
  repeat
    if ch in validch then (* checks if the character is *)
      (* hex one *)
        begin (* 7a *)
          adr1 := first;
          check (adr1, correction, number);
          memory[adr1][index] := ch;
          segment[number].changed := true;
        end (* 7a *)
      end
  until (* 7a *)

procedure FILL(var correct: boolean);
var
  i, first, second: integer;
  word: packed array [0..3] of char;
  ch: char;
  adr1, correction, number : integer;
begin (*1*)
  while j <= second do begin (*3*)
    adr1 := j;
    check(adr1, correction, number);
    memory[adr1][0] := word[0];
    memory[adr1][1] := word[1]; (* transfer each word in two memory *)
    segment[number].changed := true;
    adr1 := j + 1;
    check (adr1, correction, number);
  end
end

```

```

1561      memory[adr1][0] := word[2];
1562      memory[adr1][1] := word[3];
1563      segment[number].changed := true;
1564      j := j + 2;
1565      end; (*3*)
1566
1567      if odd(first) or odd(second) then begin (*4*)
1568          er; (* prints an error message *)
1569          correct := true (* in order to prevent the MONITOR to *)
1570              (* print also another error message *)
1571          end (*4*)
1572          end; (* 1 FILL *)
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605

```

```

procedure MOVE(var correct: boolean);

var
    j, first, second, number: integer;
    adr1, adr2, cor1, cor2, num1, num2 : integer;

begin (*1*)
    firstsecond (first, second, correct);
    if correct then (* correct and the two addresses *)
        readaddress(number, correct); (* number of bytes for transfer *)
    j := 0;
    while correct and (j <= number) do begin (*2*)
        if (first + j <= maxvirtual) and (second + j <= maxvirtual) then
            adr1 := first + j;
            adr2 := second + j;
            check (adr1, cor1, num1 );
            check (adr2, cor2, num2);

```

```
segment(num?).changed := true;  
  j := j + 1;  
  end (*2*)  
end; (* 1 MOVF *)
```

1626  
1627  
1628  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649  
1650

# APPENDIX D TEST PROGRAMS SOURCE LISTINGS

test3 Page 1 Mon Mar 30 08:42:43 1981

```

1
2
3
4
5
6
7
8 : This test checks all the following types of opcodes :
9 : - COMFIG
10 : - RESFLG
11 : - SETFLG
12 : - CLR (B)
13 : - NEG (B)
14 : - TEST (R,L)
15 : - COM (B)
16 : - EXTS (R,I)
17 : - RIT (B)
18 : - RES (B)
19 : - SET (B)
20 : - TSFI (R)
21 :
22 :
23 :
24 :
25 2102 0422;
26 2101 0422;
27 80F5;
28 80F3;
29 8011;
30 8C18;
31 8C98;
32 8028;
33 8007;
34 4005 0400 1234;
35 4005 0402 1111;
36 4005 0404 0660;
37 4005 0406 F00F;
38 2101 0400;
39 0C18;
40
41
42
43
LD R2 ← 0422 H
LD R1 ← 0422 H
COMFLG FLAGS (4:7) XOR INSTRUCTION (4:7)
RESFLG FLAGS(4:7) AND NOT INSTRUCTION (4:7)
SETFLG FLAGS (4:7) OR INSTRUCTION (4:7)
CLRB RH1
CLRB RL1
CLR R2
NOP
STORE 1234 H , MEM (0400)
STORE 1111 H , MEM (0402)
STORE 0660 H , MEM (0404)
STORE F00F H , MEM (0406)
LD R1 ← 0400 H
CLRB , MEM(R1)

```



```

CLR , MEM ((R1))
LD R2 ← MEM ((R1))
CLRB MEM (0402)
LD R2 ← MFM (0402)
CLR MEM (0402)
LD R2 ← MEM (0402)
LD R2 ← 0004 H
CLR MEM (0400 + ((R2)))
LD R2 ← MEM (0404)
NEGB RH1
NEGB RL1
NEG R1
NEG MEM (0406)
LD R2 ← MEM (0406)
STORF F00F H , MEM(0408)
LD R2 ← 0408 H
NEG MEM ((R2))
LD R3 ← MEM (0408)
NEG MEM ((R1) + 0008)
LD R3 ← MEM (0408)
TESTL R2 OR 0 (RESET S FLAG)
TEST R3 OR 0 (SET S FLAG)
TESTB RH3 OR 0 (SET PV,S FLAGS)
TESTL MEM((R2)) OR 0 (SET S)
TEST MFM ((R2) OR 0 (SET S FLAG)
TESTL MEM (0408)OR 0 (SET S FLAG)
TEST MFM (0408) OR 0 (SET S FLAG)
TESTB MEM (0408) OR 0 (SET S, PV)
TESTL MEM ((R2) + 0000)
TEST MFM ((R2) + 0000)
TESTB MEM ((R2) + 0000)
LD R3 ← F00F H
CUMB RH3
CUM R3

```

```

66 0D18;
67 2112;
68 4C08 0402;
69 6102 0402;
70 4D08 0402;
71 6102 0402;
72 2102 0004;
73 4D28 0400;
74 6102 0404;
75 8C12;
76 8C92;
77 8D12;
78 4D02 0406;
79 6102 0406;
80 4D05 0408 F00F;
81 2102 0408;
82 0D22;
83 6103 0408;
84 4D12 0008;
85 6103 0408;
86 9C20;
87 8D34;
88 8C34;
89 1C20;
90 0D24;
91 5C00 0408;
92 4D04 0408;
93 4C04 0408;
94 5C20 0000;
95 4D24 0000;
96 4C24 0000;
97 2103 F00F;
98 8C30;
99 8D30;
100
101
102
103
104
105
106
107
108

```

```
121
122
123
124
125 6104 0408;
126 0D20;
127 6104 0408;
128 4C00 0408;
129 6104 0408;
130 4D00 0408;
131 4D20 0000;
132 6104 0408;
133 2107 00F0;
134 RIF0;
135 B17A;
136 R167;
137 2101 0400;
138 2102 0002;
139 0D15 1234;
140 A6F2;
141 A77A;
142 2712;
143 2612;
144 6703 0400;
145 6603 0400;
146 6714 0000;
147 2702 0A00;
148 A262;
149 A362;
150 6104 0400;
151 2312;
152 2214;
153 6304 0400;
154 6201 0400;
155 6315 0000;
156 6105 0400;
157 2502 0700;
158 0D15 0000;
159 A5B1;
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

LD R4 ← MEM (0408)  
CUM MEM ((R2))  
LD R4 ← MEM (0408)  
COMB MEM (0408)  
LD R4 ← MEM (0408)  
CUM MEM (0408)  
CUM MEM ((R2) + 0000)  
LD R4 ← MEM (0408)  
LD R7 ← 00F0 H  
EXTSR RL7  
FXTS R7  
EXTL RR6  
LD R1 ← 0400H  
LD R2 ← 0002 H  
STORE 1234 H MEM ((R1))  
R1B Z ← NOT R17 (2)  
R1T Z ← NOT R7 (10)  
R1T Z ← NOT MEM ((R1))(2)  
R1B Z ← NOT MEM ((R1))(2)  
R1T Z ← NOT MEM (0400)(3)  
R1B Z ← NOT MEM (0400)(3)  
R1T Z ← NOT MEM ((R1) + 0000)(4)  
R1T Z ← NOT R10((R2))  
RESB BIT RH6 (2)  
RES R1T R6 (2)  
LD R4 ← MEM (0400)  
RES R1T MEM ((R1))(2)  
RESB BIT MEM ((R1))(4)  
RES R1T MEM (0400)(4)  
RESB BIT MEM ((R1) + 0000)(1)  
RES BIT MEM ((R1) + 0000)(5)  
LD R5 ← MEM (0400)  
SET BIT R7 (R2)  
STORE 0000 H, MEM ((R1))  
SET BIT (R11)(1)

|     |                 |                                     |
|-----|-----------------|-------------------------------------|
| 186 | 2510;           | SET BIT MEM((R1)(0){0001})          |
| 187 | 2410;           | SETB BIT MEM ((R1))(0){0101}        |
| 188 | 6501 0400;      | SET BIT MEM(0400)(1){0103}          |
| 189 | 6401 0400;      | SETB BIT MEM (0400)(1){0303}        |
| 190 | 6512 0000;      | SET BIT MEM (0400 + 0000) (2){0507} |
| 191 | 6412 0000;      | SETB BIT MEM(0400 + 0000) (2){0707} |
| 192 | 6104 0400;      | LD R4 ← MEM (0400)                  |
| 193 | 2105 8000;      | LD R5 ← R000 H                      |
| 194 | 2103 8000;      | LD R3 ← 8000 H                      |
| 195 | 8056;           | TSFT S ← R5 (15)                    |
| 196 | ;               | R6 ← 111...11                       |
| 197 | 8C36;           | TSEIR S ← RH3(7)                    |
| 198 | ;               | RH3 ← 1...1                         |
| 199 | 4D05 0400 8000; | STORE R000 H MEM (0400)             |
| 200 | 0C16;           | TSEIR MEM(R1)                       |
| 201 | 6104 0400;      | LD R4 ← MEM (0400)                  |
| 202 | 0D16;           | TSFT MEM (R1)                       |
| 203 | 6104 0400;      | LD R4 ← MEM (0400)                  |
| 204 | 4D05 0460 8000; | SINRF R000 H , MFM (04600)          |
| 205 | 4D16 0060;      | TSFT MEM ((R1) + 0060))             |
| 206 | 6104 0460;      | LD R4 ← MEM (0460)                  |
| 207 | 4D05 0400 8000; | STORE R000 H MEM (0400)             |
| 208 | 4C06 0400;      | TSEIR MEM (0400)                    |
| 209 | 6104 0400;      | LD R4 ← MEM (0400)                  |
| 210 | 4D06 0400;      | TSFT MEM (0400)                     |
| 211 | 6104 0400;      | LD R4 ← MEM (0400)                  |
| 212 | AFC8;           | IF CC IS TRUE THEN R(12)(0) ← 1     |
| 213 | AE18;           | IF CC IS TRUE THEN RH1(0) ← 1       |
| 214 | 7C04;           | F1 (V1 , NV1)                       |
| 215 | 7C00;           | DI (V1 , NV1)                       |
| 216 | 7A00;           |                                     |
| 217 |                 |                                     |
| 218 |                 |                                     |
| 219 |                 |                                     |
| 220 |                 |                                     |
| 221 |                 |                                     |
| 222 |                 |                                     |
| 223 |                 |                                     |
| 224 |                 |                                     |
| 225 |                 |                                     |
| 226 |                 |                                     |
| 227 |                 |                                     |
| 228 |                 |                                     |

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41

OUTPUT OF TEST No 3.

-----

The following list is the output of the test program No 3 that has been executed in the TRACE mode. In the output comments have been inserted to clarify the results of the execution of each op code. The comments are referred to the change of the data and the flags of the RFC register.

START

-----

```
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0000 0000 0000 0000 0900 0000 4000
** PC = 0 ** OP CODE = 33
{ LD R2, 0422 THE REGISTER R2 BECOMES 0422 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0000 0422 0000 0000 0000 0000 0000 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0000 0000 0000 0000 0900 0000 4000
```

```

66 { LD R1, 0422 THE REGISTER R1 BECOMES 0422
67 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
68 000 0422 0422 0000 0000 0000 0000 0000 0000 0000 0000
69 R11---R12---R13---R14---R15---RPC---RFC---
70 0000 0000 0000 0000 0900 0008 4000
71 ** PC = 8 ** UPCODE = 141
72 { CUMFLG FLAGS (4:7) XOR INSTRUCTION (4:7)
73 0 XOR F EQUALS F SO THE RFC REGISTER BECOMES 40F0 }
74 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
75 0000 0422 0422 0000 0000 0000 0000 0000 0000 0000 0000
76 R11---R12---R13---R14---R15---RPC---RFC---
77 0000 0000 0000 0000 0900 000A 40F0
78 ** PC = 10 ** UPCODE = 141
79 { RESFLG FLAGS (4:7) AND NOT INSTRUCTION (4:7)
80 F AND NOT F EQUALS 0 SO THE RFC REGISTER BECOMES 4000 }
81 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
82 0000 0422 0422 0000 0000 0000 0000 0000 0000 0000 0000
83 R11---R12---R13---R14---R15---RPC---RFC---
84 0000 0000 0000 0000 0900 000C 4000
85 ** PC = 12 ** UPCODE = 141
86 { SETFLG FLAGS (4:7) OR INSTRUCTION (4:7)
87 0 OR 1 EQUALS 1 SO THE RFC REGISTER BECOMES 4010 }
88 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
89 0000 0422 0422 0000 0000 0000 0000 0000 0000 0000 0000
90 R11---R12---R13---R14---R15---RPC---RFC---
91 0000 0000 0000 0000 0900 000E 4010
92 ** PC = 14 ** UPCODE = 140
93 { CLRB RH1, REGISTER RH1 BECOMES 00 }
94 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
95 0000 0022 0422 0000 0000 0000 0000 0000 0000 0000 0000
96 R11---R12---R13---R14---R15---RPC---RFC---
97 0000 0000 0000 0000 0900 0010 4010
98 ** PC = 16 ** UPCODE = 140
99 { CLRB RL1, REGISTER RL1 BECOMES 00 }

```

```

100
101
102
103
104
105
106
107
108

```



```

121 R0--R1--R2--R3--R4--R5--R6--R7--R8--R9--R10---
122 . 0000 0000 0422 0000 0000 0000 0000 0000 0000 0000 0000
123 R11--R12--R13--R14--R15--RPC--RFC--
124 0000 0000 0000 0000 0900 0012 4010
125 ** PC = 18 * UPCODE = 141
126 { CLR R2 , REGISTER R2 BECOMES 0000 }
127 R0--R1--R2--R3--R4--R5--R6--R7--R8--R9--R10---
128 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
129 R11--R12--R13--R14--R15--RPC--RFC--
130 0000 0000 0000 0000 0900 0014 4010
131 ** PC = 20 * UPCODE = 141
132 { NOP , DO NOTHING }
133 R0--R1--R2--R3--R4--R5--R6--R7--R8--R9--R10---
134 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
135 R11--R12--R13--R14--R15--RPC--RFC--
136 0000 0000 0000 0000 0900 0016 4010
137 ** PC = 22 * UPCODE = 77
138 { STORE 1234 H IN MEMORY ( 0400 ) }
139 R0--R1--R2--R3--R4--R5--R6--R7--R8--R9--R10---
140 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
141 R11--R12--R13--R14--R15--RPC--RFC--
142 0000 0000 0000 0000 0900 001C 4010
143 ** PC = 28 * UPCODE = 77
144 { STORE 1111 H IN MEMORY ( 0402 ) }
145 R0--R1--R2--R3--R4--R5--R6--R7--R8--R9--R10---
146 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
147 R11--R12--R13--R14--R15--RPC--RFC--
148 0000 0000 0000 0000 0900 0022 4010
149 { STORE 0600 H IN MEMORY ( 0404 ) }
150 ** PC = 34 * UPCODE = 77
151 R0--R1--R2--R3--R4--R5--R6--R7--R8--R9--R10---
152 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
153 R11--R12--R13--R14--R15--RPC--RFC--
154 0000 0000 0000 0000 0900 0028 4010
155 { STORE F00F IN MEMORY ( 0406 ) }
156 R0--R1--R2--R3--R4--R5--R6--R7--R8--R9--R10---
157 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
158 R11--R12--R13--R14--R15--RPC--RFC--
159 0000 0000 0000 0000 0900 0028 4010

```

```

R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFC--
0000 0000 0000 0000 0900 002E 4010
** PC = 4b ** UPCODE = 33
{ LD R1 , 0400 , REGISTER R1 BECOMES 0400 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0000 0000 0000 0000 0000 0000 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFC--
0000 0000 0000 0000 0900 0032 4010
** PC = 50 ** UPCODE = 12
{ CLR8 MEMORY (R1) , MEMORY(R1) FROM 1234 BECOMES 0034 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0000 0000 0000 0000 0000 0000 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFC--
0000 0000 0000 0000 0900 0034 4010
** PC = 52 ** UPCODE = 32
{ LD R#2 MEMORY (R1) SO REGISTER R#2 REMAINS 00 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0000 0000 0000 0000 0000 0000 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFC--
0000 0000 0000 0000 0900 0036 4010
** PC = 54 ** UPCODE = 13
{ CLR MEMORY (R1) SO MEMORY(R1) FROM 0034 BECOMES 0000 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0000 0000 0000 0000 0000 0000 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFC--
0000 0000 0000 0000 0900 0038 4010
** PC = 5b ** UPCODE = 33
{ LD R#2 MEMORY (R1) SO R#2 REMAINS 0000 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0000 0000 0000 0000 0000 0000 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFC--
0000 0000 0000 0000 0900 003A 4010

```



```

241
242
243
244
245 * PC = 58 ** UPCODE = 76
246 { CLR MEMORY (0402) SO MEMORY (0402) FROM 1111 BECOMES 0011 }
247 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
248 0000 0400 0000 0000 0000 0000 0000 0000 0000 0000 0000
249 R11---R12---R13---R14---R15---RPC---RFC---
250 0000 0000 0000 0000 0900 003E 4010
251 * PC = 62 ** UPCODE = 97
252 { LD R2 MEMORY(0402) SO R2 BECOMES 0011 }
253 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
254 0000 0400 0011 0000 0000 0000 0000 0000 0000 0000 0000
255 R11---R12---R13---R14---R15---RPC---RFC---
256 0000 0000 0000 0000 0900 0042 4010
257 * PC = 66 ** UPCODE = 77
258 { CLR MEMORY(0402) SO MEMORY (0402) FROM 0011 BECOMES 0000 }
259 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
260 0000 0400 0011 0000 0000 0000 0000 0000 0000 0000 0000
261 R11---R12---R13---R14---R15---RPC---RFC---
262 0000 0000 0000 0000 0900 0046 4010
263 * PC = 70 ** UPCODE = 97
264 { LD R2 MEMORY(0402) SO R2 BECOMES 0000 }
265 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
266 0000 0400 0000 0000 0000 0000 0000 0000 0000 0000 0000
267 R11---R12---R13---R14---R15---RPC---RFC---
268 0000 0000 0000 0000 0900 004A 4010
269 * PC = 74 ** UPCODE = 33
270 { LD R2 , 0004 SO R2 BECOMES 0004 }
271 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
272 0000 0400 0004 0000 0000 0000 0000 0000 0000 0000 0000
273 R11---R12---R13---R14---R15---RPC---RFC---
274 0000 0000 0000 0000 0900 004F 4010
275 * PC = 78 ** UPCODE = 77
276 { CLR MEMORY ( 0400 + R2) SO MEMORY ( 0404) FROM 0660 BECOMES
277 0000 }
278 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
279 0000 0400 0004 0000 0000 0000 0000 0000 0000 0000 0000
280
281
282
283

```

```

306 0000 0000 0000 0900 0052 4010
307 ** PC = 82 ** OPCODE = 97
308 { LD R2 MEMORY( 0404) SO R2 BECOMES 0000 }
309 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
310 0000 0400 0000 0000 0000 0000 0000 0000 0000 0000 0000
311 R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
312 0000 0000 0000 0000 0900 0056 4010
313 ** PC = 86 ** OPCODE = 140
314 { NEG R1, R1 BECOMES FC AND THE FLAGS C AND S ARE SET .
315 SU RFC BECOMES 40A0 }
316 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
317 0000 FC00 0000 0000 0000 0000 0000 0000 0000 0000 0000
318 R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
319 0000 0000 0000 0000 0900 0058 40A0
320 ** PC = 88 ** OPCODE = 140
321 { NEG R1, R1 REMAINS 00 AND THE Z FLAG IS RESET.
322 SU RFC BECOMES 40A0 }
323 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
324 0000 FC00 0000 0000 0000 0000 0000 0000 0000 0000 0000
325 R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
326 0000 0000 0000 0000 0900 005A 40A0
327 ** PC = 90 ** OPCODE = 141
328 { NEG R1, R1 BECOMES 0400, Z FLAG IS RESET AND THE C FLAG IS SET.
329 SU RFC BECOMES 4080 }
330 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
331 0000 0400 0000 0000 0000 0000 0000 0000 0000 0000 0000
332 R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
333 0000 0000 0000 0000 0900 005C 4080
334 ** PC = 92 ** OPCODE = 77
335 { NEG MEMORY ( 0406 ), MEMORY (0406) FROM FOUR BECOMES OFF1,
336 AND THE C FLAG IS SET. SO RFC BECOMES 4080 }
337 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
338 0000 0400 0000 0000 0000 0000 0000 0000 0000 0000 0000
339 R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---

```

340  
341  
342  
343  
344  
345  
346  
347  
348

```

361
362
363
364
365
366      0000 0000 0000 0900 0060 4080
367      ** PC = 96 ** UPCODE = 97
368      { LD R2 , MEMORY ( 0406 ) SO R2 BECOMES OFF1 }
369      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
370      0000 0400 00FF 0000 0000 0000 0000 0000 0000 0000 0000
371      R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
372      0000 0000 0000 0000 0900 0064 4080
373      { STORE F00F IN MEMORY ( 0408 ) }
374      ** PC = 100 ** UPCODE = 77
375      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
376      0000 0400 00FF 0000 0000 0000 0000 0000 0000 0000 0000
377      R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
378      0000 0000 0000 0000 0900 006A 4080
379      { LD R2 , 0408 SO R2 BECOMES 0408 }
380      ** PC = 106 ** UPCODE = 33
381      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
382      0000 0400 0408 0000 0000 0000 0000 0000 0000 0000 0000
383      R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
384      0000 0000 0000 0000 0900 006E 4080
385      ** PC = 110 ** UPCODE = 13
386      { NEG MEMORY ( R2 ) , MEMORY ( 0408 ) FROM F00F BECOMES OFF1 ,
387      AND C FLAG IS SET.SU RFC BECOMES 4080 }
388      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
389      0000 0400 0408 0000 0000 0000 0000 0000 0000 0000 0000
390      R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
391      0000 0000 0000 0000 0900 0070 4080
392      ** PC = 112 ** UPCODE = 97
393      { LD R3 MEMORY ( 040A ) , SU R3 BECOMES OFF1 }
394      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
395      0000 0400 0408 00FF 0000 0000 0000 0000 0000 0000 0000
396      R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
397      0000 0000 0000 0000 0900 0074 4080
398      ** PC = 116 ** UPCODE = 77
399      { NEG MEMORY ( R1 + 0008 ) , MEMORY ( 0408 ) FROM OFF1 BECOMES
400      F00F , S AND C FLAGS ARE SET. SU RFC BECOMES 40A0 }
401
402

```

```

0000 0400 0408 00F1 0000 0000 0000 0000 0000 0000 0000 0000
R11--K12--R13--R14--R15--RPC--R10---
0000 0000 0000 0000 0000 0078 40A0
** PC = 120 ** OPNDF = 97
{ LD R3, MEMORY ( 0408 ), SO R3 BECOMES 00F }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0408 00F 0000 0000 0000 0000 0000 0000 0000 0000
R11--K12--R13--K14--R15--RPC--R10---
0000 0000 0000 0000 0900 007C 40A0
** PC = 124 ** OPNDF = 156
{ TEST R2 OR 0 , CLEARS THE S FLAG SO RFC BECOMES 4080 }

{ NOTE : ALL THE IFST OPERATIONS DO NOT AFFECT THE C FLAG
  SO THE C FLAG WILL REMAIN IN SET CONDITION }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0408 00F 0000 0000 0000 0000 0000 0000 0000 0000
R11--R12--R13--K14--R15--RPC--R10---
0000 0000 0000 0000 0900 007F 4080
** PC = 126 ** OPNDF = 141
{ TEST R3 OR 0 SETS THE S FLAG SO THE RFC REMAINS 40A0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0408 00F 0000 0000 0000 0000 0000 0000 0000 0000
R11--R12--R13--K14--R15--RPC--R10---
0000 0000 0000 0000 0900 0080 40A0
** PC = 128 ** OPNDF = 140
{ TEST R3 OR 0 , SETS THE S FLAG AND THE P FLAG SO
  RFC BECOMES 40B0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0408 00F 0000 0000 0000 0000 0000 0000 0000 0000
R11--R12--R13--K14--R15--RPC--R10---
0000 0000 0000 0000 0900 0082 40B0
** PC = 130 ** OPNDF = 28
{ TEST MEMORY (R2) OR 0 , MEMORY ( 0408 ) EQUALS 00F0000 SO
  SETS THE S FLAG AND RFC REMAINS 40B0 }

```



```

481 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---P10---
482 0000 0400 0408 F00F 0000 0000 0000 0000 0000 0000 0000
483 R11---R12---R13---R14---R15---RPC---RFC---
484 0000 0000 0000 0000 0900 0084 40R0
485 ** PC = 132 ** UPCODE = 13
486 { TEST MEMORY (0408) OR 0, MEMORY (0408) EQUALS F00F S0
487 SETS THE S FLAG AND RFC REMAINS 40R0 }
488 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
489 0000 0400 0408 F00F 0000 0000 0000 0000 0000 0000 0000
490 R11---R12---R13---R14---R15---RPC---RFC---
491 0000 0000 0000 0000 0900 0086 40R0
492 ** PC = 134 ** UPCODE = 92
493 { TEST MEMORY (0408) SO RFC REMAINS 40B0 }
494 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
495 0000 0400 0408 F00F 0000 0000 0000 0000 0000 0000 0000
496 R11---R12---R13---R14---R15---RPC---RFC---
497 0000 0000 0000 0000 0900 008A 40R0
498 ** PC = 138 ** UPCODE = 77
499 { TEST MEMORY ( 0408) SO RFC REMAINS 40B0 }
500 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
501 0000 0400 0408 F00F 0000 0000 0000 0000 0000 0000 0000
502 R11---R12---R13---R14---R15---RPC---RFC---
503 0000 0000 0000 0000 0900 008E 40R0
504 ** PC = 142 ** UPCODE = 76
505 { TEST MEMORY (0408) OR 0, MEMORY (0408) EQUALS F0 S0
506 SETS THE S AND P FLAG AND RFC REMAINS 40B0 }
507 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
508 0000 0400 0408 F00F 0000 0000 0000 0000 0000 0000 0000
509 R11---R12---R13---R14---R15---RPC---RFC---
510 0000 0000 0000 0000 0900 0092 40H0
511 ** PC = 146 ** UPCODE = 92
512 { TEST MEMORY (R2 + 0000) OR 0, R2 + 0000 EQUALS 0408
513 SU RFC REMAINS 40B0 }
514 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
515 0000 0400 0408 F00F 0000 0000 0000 0000 0000 0000 0000
516
517
518
519
520
521
522
523

```

```

0000 0000 0000 0000 0900 0096 40H0
** PC = 150 ** UPCODE = 77
{ TEST MEMORY ( R2 + 0000 ) UR 0 , SU RFC REMAINS 40R0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0408 F00F 0000 0000 0000 0000 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0000 0000 0000 0000 0900 009A 40H0
** PC = 154 ** UPCODE = 76
{ TEST MEMORY ( R2 + 0000 ) , SU RFC REMAINS 40B0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0408 F00F 0000 0000 0000 0000 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0000 0000 0000 0000 0900 009F 40H0
** PC = 158 ** UPCODE = 33
{ LD R3 , FF0F SO R3 BFCOMES FF0F }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0408 FF0F 0000 0000 0000 0000 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0000 0000 0000 0000 0900 00A2 40R0
** PC = 162 ** UPCODE = 140
{ COMR RH3 , RH3 FROM FF RECUMES 00 , Z AND PV FLAGS ARE SET.
  SO RFC RECUMES 40D0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0408 000F 0000 0000 0000 0000 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0000 0000 0000 0000 0900 00A4 40D0
** PC = 164 ** UPCODE = 141
{ COM R3 , R3 FROM 000F BFCOMES FFF0 AND S FLAG IS SET.
  SU RFC BFCOMES 40R0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0408 FFF0 0000 0000 0000 0000 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0000 0000 0000 0000 0900 00A6 40R0
** PC = 166 ** UPCODE = 97

```

```

601      { LD R4 , 0408 .R2 BFCOMES 0408 }
602      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
603      0000 0400 0408 FFF0 F00F 0000 0000 0000 0000 0000 0000 0000
604      R11---R12---R13---R14---R15---RPC---RFC---
605      0000 0000 0000 0000 0900 00AA 40B0
606      * PC = 170 * * UPCODE = 13
607      { CUM MEMORY ( R2 ) .MEMORY ( 0408 ) FROM F00F BECOMES 0FF0 ,
608      AND S FLAG IS RESET.S0 RFC REMAINS 4090 }
609      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
610      0000 0400 0408 FFF0 F00F 0000 0000 0000 0000 0000 0000 0000
611      R11---R12---R13---R14---R15---RPC---RFC---
612      0000 0000 0000 0000 0900 00AC 4090
613      * PC = 172 * * UPCODE = 97
614      { LD R4 , MEMORY ( 0408 ) , S0 R4 BECOMES 0FF0 }
615      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
616      0000 0400 0408 FFF0 0FF0 0000 0000 0000 0000 0000 0000 0000
617      R11---R12---R13---R14---R15---RPC---RFC---
618      0000 0000 0000 0000 0900 00B0 4090
619      * PC = 176 * * UPCODE = 76
620      { CUMB MEMORY ( 0408 ) , MEMORY ( 0408 ) FROM 0FF0 BFCOMES
621      F0F0 AND S AND P FLAGS ARE SET.S0 RFC BECOMES 40B0 }
622      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
623      0000 0400 0408 FFF0 0FF0 0000 0000 0000 0000 0000 0000 0000
624      R11---R12---R13---R14---R15---RPC---RFC---
625      0000 0000 0000 0000 0900 00B4 40B0
626      * PC = 180 * * UPCODE = 97
627      { LD R4 , MEMORY ( 0408 ) S0 R4 BECOMES F0F0 }
628      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
629      0000 0400 0408 FFF0 F0F0 0000 0000 0000 0000 0000 0000 0000
630      R11---R12---R13---R14---R15---RPC---RFC---
631      0000 0000 0000 0000 0900 00B8 40B0
632      * PC = 184 * * UPCODE = 77
633      { CUM MEMORY ( 0408 ) . MEMORY ( 0408 ) FROM F0F0 BECOMES
634      0F0F , AND S FLAG IS RESET S0 RFC BECOMES 4090 }
635      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---

```



```

R11--R12--R13--R14--R15--RPC--RFC--
0000 0000 0000 0000 0900 00BC 4090
** PC = 188 ** OPCLDF = 77
{ COM MEMORY ( R2 + 0000 ) . MEMORY ( 0408 ) FROM 0F0F
BECOMES F0F0 AND THE S FLAG IS SET .SO RFC BECOMES 40B0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0408 FFF0 F0F0 0000 0000 0000 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFC--
0000 0000 0000 0000 0900 00C0 40B0
** PC = 192 ** OPCLDF = 97
{ LD R4 MEMORY ( 0408 ) SO R4 BECOMES F0F0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0408 FFF0 F0F0 0000 0000 0000 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFC--
0000 0000 0000 0000 0900 00C4 40B0
** PC = 196 ** OPCLDF = 35
{ LD R7 , 00F0 SO R7 BECOMES 00F0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0408 FFF0 F0F0 0000 0000 00F0 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFC--
0000 0000 0000 0000 0900 00C8 40B0
** PC = 200 ** OPCLDF = 177
{ EX1SH RL7 SU R7 FROM 00F0 BECOMES FFF0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0408 FFF0 F0F0 0000 0000 FFF0 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFC--
0000 0000 0000 0000 0900 00CA 40B0
** PC = 202 ** OPCLDF = 177
{ EX1S R7 SU R6 FROM 0000 FFF0 BECOMES FFFF FFF0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0408 FFF0 F0F0 0000 FFFF FFF0 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFC--
0000 0000 0000 0000 0900 00CC 40B0
** PC = 204 ** OPCLDF = 177

```



```

R11--R12--R13--R14--R15--R16--R17--R18--R19--R20--
0000 0000 0000 0000 0900 0000 0000 0000 0000 0000
** PC = 222 ** * PC = 39
{ BIT 2, NOT MEMORY ( R1 ) ( 2 ), MEMORY ( R1 ) EQUALS 1234
  SO RESET Z FLAG AND RFC REMAINS 40B0 }
R0--R1--R2--R3--R4--R5--R6--R7--R8--R9--R10--R11--R12--R13--R14--R15--R16--R17--R18--R19--R20--
0000 0400 0002 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
R11--R12--R13--R14--R15--R16--R17--R18--R19--R20--
0000 0000 0000 0000 0900 00E0 40B0
** PC = 224 ** * PC = 3B
{ BIT 2, MEMORY ( R1 ) ( 2 ), MEMORY ( R1 ) EQUALS 12 SU
  SEIS THE 7 FLAG AND RFC BECOMES 40F0 }
R0--R1--R2--R3--R4--R5--R6--R7--R8--R9--R10--R11--R12--R13--R14--R15--R16--R17--R18--R19--R20--
0000 0400 0002 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
R11--R12--R13--R14--R15--R16--R17--R18--R19--R20--
0000 0000 0000 0000 0900 00E2 40F0
** PC = 226 ** * PC = 103
{ BIT 2, MEMORY ( 0400 ) ( 3 ), SO IT SEIS THE 7 FLAG AND
  RFC BECOMES 40F0 }
R0--R1--R2--R3--R4--R5--R6--R7--R8--R9--R10--R11--R12--R13--R14--R15--R16--R17--R18--R19--R20--
0000 0400 0002 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
R11--R12--R13--R14--R15--R16--R17--R18--R19--R20--
0000 0000 0000 0000 0900 00E6 40F0
** PC = 230 ** * PC = 102
{ BIT 2, MEMORY ( 0400 ) ( 2 ), SO IT SEIS THE 7 FLAG AND
  RFC BECOMES 40F0 }
R0--R1--R2--R3--R4--R5--R6--R7--R8--R9--R10--R11--R12--R13--R14--R15--R16--R17--R18--R19--R20--
0000 0400 0002 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
R11--R12--R13--R14--R15--R16--R17--R18--R19--R20--
0000 0000 0000 0000 0900 00EA 40F0
** PC = 234 ** * PC = 103
{ BIT 2, MEMORY ( R1 + 0000 ) ( 4 ), SO RESET Z FLAG AND
  RFC BECOMES 40B0 }
R0--R1--R2--R3--R4--R5--R6--R7--R8--R9--R10--R11--R12--R13--R14--R15--R16--R17--R18--R19--R20--

```

```

786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828

```



```

906 { RESB BIT MEMORY ( 0400 ) ( 4 ) , MEMORY ( 0400 ) FROM
907 1230 BFCOMFS 0230 }
908 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
909 0000 0400 0002 FFF0 1234 FFFF FBFB FFF0 0000 0000 0000
910 R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
911 0000 0000 0000 0000 0900.00FF 40F0
912 ** PC = 254 ** UPCODE = 99
913 { RES BIT MEMORY ( 0400 ) ( 4 ) , MEMURUY ( 0400 ) FROM
914 0230 BFCOMFS 0220 }
915 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
916 0000 0400 0002 FFF0 1234 FFFF FBFB FFF0 0000 0000 0000
917 R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
918 0000 0000 0000 0000 0900 0102 40F0
919 ** PC = 258 ** UPCODE = 98
920 { RES BIT MEMORY ( R1 + 0000 ) ( 4 ) , MEMORY ( 0400 )
921 FROM 0220 RECUMES 0020 }
922 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
923 0000 0400 0002 FFF0 1234 FFFF FBFB FFF0 0000 0000 0000
924 R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
925 0000 0000 0000 0000 0900 0106 40F0
926 ** PC = 262 ** UPCODE = 99
927 { RES BIT MEMORY ( R1 + 0000 ) ( 5 ) , MEMORY ( 0400 ) FROM
928 0020 BFCOMFS 0000 }
929 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
930 0000 0400 0002 FFF0 1234 FFFF FBFB FFF0 0000 0000 0000
931 R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
932 0000 0000 0000 0000 0900 010A 40F0
933 ** PC = 266 ** UPCODE = 97
934 { ID R5 , MEMORY ( 0400 ) SO R5 RECUMES 0000 }
935 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
936 0000 0400 0002 FFF0 1234 0000 FBFB FFF0 0000 0000 0000
937 R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
938 0000 0000 0000 0000 0900 010F 40F0
939 ** PC = 270 ** UPCODE = 37

```



```

961      { AET BIT R7 ( R2 ) , SU R7 FROM FFF0 BECOMES FFF4 }
962      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
963      0000 0400 0002 FFF0 1234 0000 FBFB FFF4 0000 0000 0000
964      R11---R12---R13---R14---R15---RPC---RFC---
965      0000 0000 0000 0000 0900 0112 40F0
966      ** PC = 274 ** OPCODE = 13
967      { STORE 0000 , MEMORY ( R1 ) }
968      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
969      0000 0400 0002 FFF0 1234 0000 FBFB FFF4 0000 0000 0000
970      R11---R12---R13---R14---R15---RPC---RFC---
971      0000 0000 0000 0000 0900 0116 40F0
972      ** PC = 278 ** OPCODE = 165
973      { SET BIT R11 ( 1 ) , SU R11 BECOMES 0002 }
974      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
975      0000 0400 0002 FFF0 1234 0000 FBFB FFF4 0000 0000 0000
976      R11---R12---R13---R14---R15---RPC---RFC---
977      0002 0000 0000 0000 0900 0118 40F0
978      ** PC = 280 ** OPCODE = 37
979      { SET BIT R11 ( R2 ) SU R11 BECOMES 0006 }
980      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
981      0000 0400 0002 FFF0 1234 0000 FBFB FFF4 0000 0000 0000
982      R11---R12---R13---R14---R15---RPC---RFC---
983      0006 0000 0000 0000 0900 011C 40F0
984      ** PC = 284 ** OPCODE = 37
985      { SET BIT MEMORY ( R1 ) ( 0 ) , MEMORY ( 0400 ) BECOMES 0001 }
986      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
987      0000 0400 0002 FFF0 1234 0000 FBFB FFF4 0000 0000 0000
988      R11---R12---R13---R14---R15---RPC---RFC---
989      0006 0000 0000 0000 0900 011F 40F0
990      ** PC = 286 ** OPCODE = 36
991      { SET BIT MEMORY ( R1 ) ( 0 ) , SO MEMORY ( 0400 ) BECOMES 0101 }
992      R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
993      0000 0400 0002 FFF0 1234 0000 FBFB FFF4 0000 0000 0000
994      R11---R12---R13---R14---R15---RPC---RFC---
995      0006 0000 0000 0000 0900 0120 40F0
996      1000
997      1001
998      1003

```

```

{ SET BIT MEMORY ( 0400 ) ( 1 ) , SO MEMORY BECOMES 0103 }
R0---R1---R2---R3---R4---P5---R6---R7---R8---R9---R10---
000 0400 0002 FFF0 1234 0000 FBFB FFF4 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFL--
0006 0000 0000 0000 0900 0124 40F0
** PC = 292 ** OPCODE = 100
{ SFB BIT MEMORY ( 0400 ) ( 1 ) , SO MEMORY BECOMES 0303 }
R0---R1---R2---P3---R4---P5---R6---R7---R8---R9---R10---
0000 0400 0002 FFF0 1234 0000 FBFB FFF4 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFL--
0006 0000 0000 0000 0900 0128 40F0
** PC = 296 ** OPCODE = 101
{ SET BIT MEMORY ( 0400 + 0000 ) ( 2 ) , SO MEMORY BECOMES 0307 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0002 FFF0 1234 0000 FBFB FFF4 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFL--
0006 0000 0000 0000 0900 012C 40F0
** PC = 300 ** OPCODE = 100
{ SFB BIT MEMORY ( 0400 + 0000 ) ( 2 ) , MEMORY BECOMES 0707 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0002 FFF0 1234 0000 FBFB FFF4 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFL--
0006 0000 0000 0000 0900 0130 40F0
** PC = 304 ** OPCODE = 97
{ LD R4 , MEMORY ( 0400 ) SO R4 BECOMES 0707 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0002 FFF0 0707 0000 FBFB FFF4 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFL--
0006 0000 0000 0000 0900 0134 40F0
** PC = 308 ** OPCODE = 53
{ LD R5 , 8000 SO R5 BECOMES 8000 }
R0---R1---R2---R3---R4---P5---R6---R7---R8---R9---R10---
0000 0400 0002 FFF0 0707 8000 FBFB FFF4 0000 0000 0000
R11--R12--R13--R14--R15--RPC--RFL--

```

1026  
1027  
1028  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036  
1037  
1038  
1039  
1040  
1041  
1042  
1043  
1044  
1045  
1046  
1047  
1048  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059  
1060  
1061  
1062  
1063  
1064  
1065  
1066  
1067  
1068



```

1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
0006 0000 0000 0000 0900 0138 40F0
** PC = 312 ** OP CODE = 33
{ LD R3 , 8000 , SO R3 BECOMES 8000 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0002 8000 0707 8000 FFFF FFFF 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0006 0000 0000 0000 0900 013C 40F0
** PC = 316 ** OP CODE = 141
{ ISET S, R5 , IT SETS THE S FLAG AND R5 BECOMES FFFF. THE
RFC BECOMES 40F0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0002 8000 0707 FFFF FFFF FFFF 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0006 0000 0000 0000 0900 013E 40F0
** PC = 318 ** OP CODE = 140
{ ISETB RH3 (7) , IT SETS THE S FLAG AND RH3 BECOMES FF.
SO RFC REMAINS 40F0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0002 FFFF 0707 FFFF FFFF FFFF 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0006 0000 0000 0000 0900 0140 40F0
** PC = 320 ** OP CODE = 77
{ STORE 8000 , MEMORY ( 0400 ) }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0002 FFFF 0707 FFFF FFFF FFFF 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0006 0000 0000 0000 0900 0146 40F0
** PC = 326 ** OP CODE = 12
{ ISETB MEMORY ( R1 ) , IT SETS THE S FLAG , MEMORY ( 0400 )
FROM 8000 BECOMES FFFF AND RFC REMAINS 40F0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0002 FFFF 0707 FFFF FFFF FFFF 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0006 0000 0000 0000 0900 0148 40F0
** PC = 328 ** OP CODE = 97

```

```

R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0002 FF00 FF00 FFFF FBFB FFF4 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0006 0000 0000 0000 0900 014C 40F0
** PC = 332 ** OPCODE = 13
{ ISET MEMORY ( R1 ), IT SETS THE S FLAG , MEMORY ( 0400 )
FROM FF00 BECOMES FFFF AND RFC REMAINS 40F0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0002 FF00 FF00 FFFF FBFB FFF4 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0006 0000 0000 0000 0900 014E 40F0
** PC = 334 ** OPCODE = 97
{ LD R4 , MEMORY ( 0400 ) , SO R4 BECOMES FFFF }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0002 FF00 FFFF FFFF FBFB FFF4 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0006 0000 0000 0000 0900 0152 40F0
** PC = 338 ** OPCODE = 77
{ STORE R000 , MEMORY ( 0460 ) }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0002 FF00 FFFF FFFF FBFB FFF4 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0006 0000 0000 0000 0900 0158 40F0
** PC = 344 ** OPCODE = 77
{ ISET MEMORY ( R1 + 0060 ) , MEMORY ( 0460 ) BECOMES FFFF ,
IT SETS THE S FLAG AND RFC REMAINS 40F0 }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0002 FF00 FFFF FFFF FBFB FFF4 0000 0000 0000
R11---R12---R13---R14---R15---RPC---RFC---
0006 0000 0000 0000 0900 015C 40F0
** PC = 348 ** OPCODE = 97
{ LD R4 , MEMORY ( 0460 ) , SO R4 BECOMES FFFF }
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
0000 0400 0002 FF00 FFFF FFFF FBFB FFF4 0000 0000 0000

```

1146  
1147  
1148  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1160  
1161  
1162  
1163  
1164  
1165  
1166  
1167  
1168  
1169  
1170  
1171  
1172  
1173  
1174  
1175  
1176  
1177  
1178  
1179  
1180  
1181  
1182  
1183  
1184  
1185  
1186  
1187  
1188

```

1201
1202
1203
1204
1205 R11--R12--R13--R14--R15--RPC--RFC--
1206 0006 0000 0000 0000 0900 0160 40F0
1207 ** PC = 352 ** OP CODE = 77
1208 { SIOF 8000, MEMORY ( 0400 ) }
1209 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
1210 0000 0400 0002 FF00 FFFF FFFF FBFB FFF4 0000 0000 0000
1211 R11--R12--R13--R14--R15--RPC--RFC--
1212 0006 0000 0000 0000 0900 0160 40F0
1213 ** PC = 358 ** OP CODE = 76
1214 { ISETB MEMORY ( 0400 ) , MEMORY ( 0400 ) FROM 8000 BECOMES
1215 FF00, IT SETS THE S FLAG AND RFC REMAINS 40F0 }
1216 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
1217 0000 0400 0002 FF00 FFFF FFFF FBFB FFF4 0000 0000 0000
1218 R11--R12--R13--R14--R15--RPC--RFC--
1219 0006 0000 0000 0000 0900 016A 40F0
1220 ** PC = 362 ** OP CODE = 97
1221 { LD R4 , MEMORY ( 0400 ) , SO R4 BECOMES FF00 }
1222 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
1223 0000 0400 0002 FF00 FF00 FFFF FBFB FFF4 0000 0000 0000
1224 R11--R12--R13--R14--R15--RPC--RFC--
1225 0006 0000 0000 0000 0900 016E 40F0
1226 ** PC = 366 ** OP CODE = 77
1227 { ISET MEMORY ( 0400 ) , MEMORY ( 0400 ) FROM FF00 BECOMES
1228 FFFF , IT SETS THE S FLAG AND THE RFC REMAINS 40F0 }
1229 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
1230 0000 0400 0002 FF00 FF00 FFFF FBFB FFF4 0000 0000 0000
1231 R11--R12--R13--R14--R15--RPC--RFC--
1232 0006 0000 0000 0000 0900 0172 40F0
1233 ** PC = 370 ** OP CODE = 97
1234 { LD R4 , MEMORY ( 0400 ) , SO R4 BECOMES FFFF }
1235 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
1236 0000 0400 0002 FF00 FFFF FFFF FBFB FFF4 0000 0000 0000
1237 R11--R12--R13--R14--R15--RPC--RFC--
1238 0006 0000 0000 0000 0900 0176 40F0
1239 ** PC = 374 ** OP CODE = 175
1240
1241

```

```

1265 IF TRUE THEN SET R1C ( 0 ) , SU R1C BECOMES 0000
1266 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
1267 0000 0400 0002 FF00 FFFF FFFF FBFB FFFF 0000 0000 0000
1268 R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
1269 0006 0001 0000 0000 0900 0178 40F0
1270 ** PC = 376 ** OPCODE = 174
1271 { IF TRUE THEN SET R1C ( 0 ) , SU R1C BECOMES 05 }
1272 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
1273 0000 0500 0002 FF00 FFFF FFFF FBFB FFFF 0000 0000 0000
1274 R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
1275 0006 0001 0000 0000 0900 017A 40F0
1276 ** PC = 378 ** OPCODE = 124
1277 { ET SU R1C BECOMES 58F0 }
1278 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
1279 0000 0500 0002 FF00 FFFF FFFF FBFB FFFF 0000 0000 0000
1280 R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
1281 0006 0001 0000 0000 0900 017C 58F0
1282 ** PC = 380 ** OPCODE = 124
1283 { UT SU R1C BECOMES 40F0 }
1284 R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---
1285 0000 0500 0002 FF00 FFFF FFFF FBFB FFFF 0000 0000 0000
1286 R11---R12---R13---R14---R15---R16---R17---R18---R19---R20---
1287 0006 0001 0000 0000 0900 017E 40F0
1288 ** PC = 382 ** OPCODE = 122
1289 { HALT SU STOPS THE PROGRAM }
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308

```

```

1
2
3
4
5 ;
6
7 ;
8 ;
9 ;
10 ;
11 ; This test checks the virtual memory system
12 2101 00FF; LD R1 ← FF H
13 5F00 1000; CALL 1000 H
14 7A00; HALT
15 ; START AT 1000H
16 A913; INCREMENT R1 ← R1 + 4 H
17 5F00 2000; CALL 2000 H
18 9E08; RET
19 ; START AT 2000 H
20 A913; INCREMENT R1 ← R1 + 4 H
21 5E08 3000; JUMP 3000 H
22 9E08; RET
23 ; START AT 3000 H
24 A913; INC R1 ← R1 + 4 H
25 5F00 4000; CALL 4000 H
26 5E08 1006; JUMP 2006 H
27 ; START AT 4000H
28 A912; INC R1 ← R1 + 3H
29 5F00 5000; CALL 5000 H
30 9E08; RET
31 ; START AT 5000 H
32 AB1E; DECREMENT R1 ← R1 - 15 H
33 9E08; RET.
34
35
36
37
38
39
40
41
42

```



|    |               |      |      |  |  |
|----|---------------|------|------|--|--|
| 6  |               |      |      |  |  |
| 7  | 78000ASM 2.02 |      |      |  |  |
| 8  | LUC           | ORJ  | CODF |  |  |
| 9  |               |      |      |  |  |
| 10 |               |      |      |  |  |
| 11 |               |      |      |  |  |
| 12 |               |      |      |  |  |
| 13 |               |      |      |  |  |
| 14 |               |      |      |  |  |
| 15 |               |      |      |  |  |
| 16 | 0000          | 0001 | 0002 |  |  |
| 17 | 0004          | 0004 | 0009 |  |  |
| 18 | 0008          | 0006 | 0005 |  |  |
| 19 | 000C          | 0000 | 0008 |  |  |
| 20 | 0010          | 0007 | 0003 |  |  |
| 21 |               |      |      |  |  |
| 22 |               |      |      |  |  |
| 23 |               |      |      |  |  |
| 24 | 0014          |      |      |  |  |
| 25 | 0000          |      |      |  |  |
| 26 |               |      |      |  |  |
| 27 | 0000          | 4C05 | 0014 |  |  |
| 28 | 0004          | 0000 |      |  |  |
| 29 | 0006          | 8D18 |      |  |  |
| 30 |               |      |      |  |  |
| 31 | 0008          | 8R01 |      |  |  |
| 32 | 000A          | 5E07 | 0012 |  |  |
| 33 | 000E          | 5F08 | 0036 |  |  |
| 34 | 0012          | A112 |      |  |  |
| 35 | 0014          | A911 |      |  |  |
| 36 | 0016          | 6114 | 0000 |  |  |
| 37 | 001A          | 6126 | 0000 |  |  |
| 38 | 001E          | 8R64 |      |  |  |
| 39 | 0020          | 5F03 | 0032 |  |  |
| 40 |               |      |      |  |  |
| 41 |               |      |      |  |  |
| 42 |               |      |      |  |  |
| 43 |               |      |      |  |  |
| 44 |               |      |      |  |  |
| 45 |               |      |      |  |  |
| 46 |               |      |      |  |  |
| 47 |               |      |      |  |  |
| 48 |               |      |      |  |  |

```

SIMI SOURCE STATEMENT
1  BUBBLE SORT MODULE
2  $ LISTON
3  CONSTANT
4  FALSE := 0
5  TRUE  := 1
6  INTERNAL
7  LIST ARRAY(10 WORD) := {1,2,4,9,6,5,8,0,7,3}

8  SORT PROCEDURE
9  LOCAL
10 SWITCH BYTE
11 ENTRY
12 DO
13   LDB SWITCH,#FALSE ! INITIALIZE SWITCH!
14   CLR, R1 !CLEAR ARRAY POINTER I!
15   DO
16     CP R1,R0 !DONE ?!
17     IF UGE THEN EXIT FI
18   LD R2,R1 !INITIALIZE POINTER J!
19   INC R1, #2 ! J = I + 1 !
20   LD R4, LIST(R1)
21   LD R6, LIST(R2)
22   CP R4, R6 ! IF LIST(I) > LIST(J) THEN!
23   IF UGT THEN ! EXCHANGE TO BUBBLE LARGEST!

```

```

61
62
63
64
65 0024 4C05 0014 LDB SWITCH , #IRUF ;NUMBER TO TOP OF ARRAY!
66 0028 0101 LD LIST(R1) , R6
67 002A 6F16 LD LIST(R2) , R4
68 002E 6F24 FI
69 INC R1,#2 ; ADVANCE POINTER !
70 0032 A911 DD ; END NESTED DO LOOP!
71 0034 E809 CPB SWITCH , #FALSE ;TEST SWITCH!
72 0036 4C01 0014 IF EQ THEN RFI FI
73 003A 0000
74 003C 5E0F 0042
75 0040 9E08
76 0042 E8DE
77 0044
78
79
80 0044
81
82 0044 2100 0012 LD R0, #9*2 ; INITIALIZE LOOP CONTROL!
83 0048 5F00 0000 CALL SORT ; CALL SORT PROCEDURE!
84 004C 9E08 RET
85 004E
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103

```



6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48

```
; This code is the hexadecimal code of the program  
; BUBBLE.L that has been assembled in the Z8000  
; assembler
```

```
16 4C05 0414  
17 0000  
18 8018  
19 8B01  
20 5E07 0012  
21 5E08 0036  
22 A112  
23 A921  
24 6114 0400  
25 6126 0400  
26 8B64  
27 5E03 0032  
28 4C05 0414  
29 0101  
30 6F16 0400  
31 6F24 0400  
32 A911  
33 F8F9  
34 4C01 0414  
35 0000  
36 5E0E 0042  
37 9E08  
38 F80E  
39 2100 0012
```

|    |           |
|----|-----------|
| 61 |           |
| 62 |           |
| 63 |           |
| 64 |           |
| 65 | 5F00 0000 |
| 66 | 7A00      |
| 67 | 0000      |
| 68 | 0001 0002 |
| 69 | 0004 0009 |
| 70 | 0006 0005 |
| 71 | 0000 0008 |
| 72 | 0007 0003 |
| 73 | 0000      |
| 74 |           |
| 75 |           |
| 76 |           |
| 77 |           |
| 78 |           |
| 79 |           |
| 80 |           |
| 81 |           |
| 82 |           |
| 83 |           |
| 84 |           |
| 85 |           |
| 86 |           |
| 87 |           |
| 88 |           |
| 89 |           |
| 90 |           |
| 91 |           |
| 92 |           |
| 93 |           |
| 94 |           |

# OUTPUT OF IEST No 9

-----

The program was executed with the BREAK mode and so it follows the following steps:

- the initial loaded code is displayed with the command  
D 0000 40 W
- the BREAK option is enforced with the command B 0042 1  
and after each break message the contents of the array  
which is bubble sorted is displayed and also the contents  
of the registers using the commands D 0400 20 W and R.

```

0000* 4C05 0414 0000 8D18 8B01 5E07 0012 5F08 *L      ↑  ↑  *
0010* 0036 A112 A921 6114 0400 6126 0400 8B64 *6      a8 d*
0020* 5F03 0032 4C05 0414 0101 6F16 0400 6F24 *↑      0  o$*
0030* 0400 A911 E8E9 4C01 0414 0000 5F0F 0042 *      L  ↑  B*
0040* 9F08 E8DF 2100 0012 5F00 0000 7A00 0000 *      !  ←  z  *
** message : BREAK AT 0042
R0---R1---R2---R3---R4---R5---R6---R7---R8---R9---R10---

```





## APPENDIX E

### SEQUENCES OF CALLS.

In this appendix is provided a list of all the main procedures of the simulator program and the procedures that are called by each one procedure. The number under the parentheses defines the nesting level, starting with level equal to zero for the main program. The nesting terminates when the procedure calls a primitive one as 'setreg' or 'retvalreg'. For the primitives ones is provided a separate list at the end of the Appendix.

MAIN PROGRAM (0)

-----  
-----  
-----

- boot (1)

-----  
-----

- setreg (2)

-----

- MONITOR (1)

-----  
-----

- skipblanks (2)

-----

- MOVE (2)

-----

- firstsecond (3)

- readaddress (4)



- valuechar (5)
- skipblanks (5)
- FILL (2)
  - firstsecond (3)
    - readaddress (4)
      - valuechar (5)
      - skipblanks (5)
  - skipblanks (3)
- BREAK (2)
  - firstsecond (3)
    - readaddress (4)
      - valuechar (5)
      - skipblanks (5)
- JUMP (2)
  - readaddress (3)
    - valuechar (4)
    - skipblanks (4)
  - setreg (3)
  - er (3)
- DISPLAY (2)
  - displaymix (3)
    - dechex (4)

- valuechar (4)
- firstsecond (3)
  - readaddress (4)
    - valuechar (5)
    - skipblanks (5)
  - skipblanks (3)
  - er (3)
- REG (2)
  - 
  - REGISTER (3)
    - decnex (4)
      - retvalreg (5)
  - setreg (3)
  - STACKINTERCHANGE (3)
  - skipblanks (3)
- COMPARE (2)
  - 
  - firstsecond (3)
    - readaddress (4)
      - valuechar (5)
      - skipblanks (5)
  - dechex (3)
  - readaddress (3)
    - valuechar (4)
    - skipblanks (4)
- SEND (2)
  -

- readaddress (3)
  - valuechar (4)
  - skipblanks (4)

- NEXT (2)  
-----

- readaddress (3)
  - valuechar (4)
  - skipblanks (4)

- load (2)  
-----

- EXECUTE (1)  
-----  
-----

- REGISTER (2)  
-----

- dechex (3)
- retvalreg (3)

- decode (2)  
-----

- valuechar (3)

- setsrclist (2)

- map (3)

- settrap (2)  
-----

- TRAP (3)

- retvalreg (4)
- setreg (4)

- setreg (2)  
-----

- LOAD (2)
  - IMorIR (3)
  - setreg (3)
  - DAorX (3)
  - map (3)
  - retvalreg (3)
  - setmem (3)
- EX (2)
  - setlength (3)
  - retvalreg (3)
  - setreg (3)
  - DAorX (3)
  - setmem (3)
- STORE (2)
  - setsrddst (3)
  - setlength1 (3)
  - retvalreg (3)
  - DAorX (3)
  - setmem (3)
  - setreg (3)
- LOAD (2)
  - IMorIR (3)
  - setreg (3)

- DAorX (3)
- map (3)
- retvalreg (3)
- setmem (3)
- EX (2)
  - 
  - setlength1 (3)
  - retvalreg (3)
  - setreg (3)
  - DAorX (3)
  - setmem (3)
- MULTIOPER (2)
  - 
  - compl (3)
    - onetwocompl (4)
    - retvalreg (4)
    - setmem (4)
  - compare (3)
    - map (4)
    - setreg (4)
    - onetwocompl (4)
    - ADD (4)
  - neg (3)
    - onetwocompl (4)
    - retvalreg (4)
    - setmem (4)

- tset (3)
  - setmem (4)
- setlength1 (3)
- setsrddst (3)
- upperlow (3)
- retvalreg (3)
- setindex (3)
- nap (3)
- setreg (3)
- ORANDXOR (3)
- setmem (3)
- onetwocopl (3)
- LOGICAL (2)
  - 
  - setlength1 (3)
  - ORANDXOR (3)
  - setreg (3)
  - retvalreg (3)
- ARITHMETIC (2)
  - 
  - setlength1 (3)
  - IMorIR (3)
  - setreg (3)
  - ADD (3)
  - DAorX (3)
  - onetwocopl (3)



- map (3)
- retvalreg (3)
- setmem (3)
- testcc (3)
- setindex (3)
- upperlow (3)
- DIVMULT (2)
  - positive (3)
    - onetwocopl (4)
    - retvalreg (4)
  - setlength1 (3)
  - onetwocopl (3)
  - IMorIR (3)
  - setreg (3)
  - map (3)
- CALLRET (2)
  - retvalreg (3)
  - DAorX (3)
  - testcc (3)
  - map (3)
  - pushstack (3)
    - retvalreg (4)
    - error (4)
    - setmem (4)

- setreg (4)
- setreg (3)
- error (3)
- BITMAN (2)
  - resetset (3)
  - retvalreg (4)
  - setmem (4)
  - setsrddst (3)
  - setlength1 (3)
  - upperlow (3)
  - setindex (3)
  - DAorX (3)
  - map (3)
  - setreg (3)
  - retvalreg (3)
- INPUTOUTPUT (2)
  - message (3)
  - common (3)
    - setreg (4)
  - outputreg (3)
    - retvalreg (4)
  - inputreg (3)
    - message (4)
    - setreg (4)

- stleneth1 (3)
- IRET (2)
  - 
  - retvalreg (3)
  - map (3)
  - setreg (3)
  - STACKINTERCHANGE (3)
  - error (3)
- LDCTL (2)
  - 
  - exchange (3)
  - setsrclist (3)
  - retvalreg (3)
  - setreg (3)
- LPDS (2)
  - 
  - TRAP (3)
    - retvalreg (4)
    - setreg (4)
  - retvalreg (3)
  - DAorX (3)
  - map (3)
  - setreg (3)
  - STACKINTERCHANGE (3)
- TRANSLATE (2)
  - 
  - setsrclist (3)

- retvalreg (3)
- map (3)
- setmem (3)
- setreg (3)
- BLOCKTRANSFER (2)
  - 
  - setsrcdst (3)
  - setlength1 (3)
  - map (3)
  - retvalreg (3)
  - setmem (3)
  - setreg (3)
  - compare (3)
    - onetwocompl (4)
    - ADD (4)
    - testcc (4)
    - map (4)
- ROTATESHIFT (2)
  - 
  - valuechar (3)
  - shiftrightleft (3)
    - upperlow (4)
    - setindex (4)
  - setsrcdst (3)
  - setlength1 (3)
  - retvalreg (3)

- DAB (2)

-----

- retvalreg (3)

- setreg (3)

The primitives procedures retvalreg, setreg, ADD, ORANDXOR, map,

MorIR, DAorX, setmem call the following ones procedures :

- retvalreg (0)

- upperlow (1)

- setindex (1)

- error (2)

- setreg (2)

- upperlow (1)

- setindex (1)

- error (2)

- ADD (0)

- upperlow (1)

- setindex (1)

- error (2)

- setflags (1)

- ORANDXOR (0)

- upperlow (1)
- setindex (1)
  - error (2)
- setflags (1)

  

- map (0)
  - error (1)
  - valuechar (1)

  

- IMorIR (0)
  - map (1)
    - error (2)
    - valuechar (2)
  - retvalreg (1)
    - upperlow (2)
    - setindex (2)
      - error (3)

  

- DAorX (0)
  - map (1)
    - error (2)
    - valuechar (2)
  - retvalreg (1)
    - upperlow (2)
    - setindex (2)



- error (3)

- setmem (0)

- error (1)

- setsrddst (0)

- map (1)

- error (2)

- valuechar (2)

## APPENDIX G

### REQUIRED MODIFICATIONS FOR THE VM/370 SYSTEM

The program in order to be transferred to the OS VM/370 which supports the WATERLOO PASCAL it needs to be modified because the VM/370 supports EBCDIC code instead of ASCII and it does not assume as default input output the terminal, so the terminal is used as any other file.

The below listed modifications follow the order that are encountered in the program. In the case that the structure of the algorithm is changed then it is provided the new one.

The changes are the followings :

#### - At constant declarations

const

the following constants to be changed :

maxmem = 65535; (\* increases the array of the memory  
at the 64 k \*)

psaarea = 56320; (\* default value for the PS area \*)

systemstackpointer = 61440; (\* default value for the  
system stackpointer \*)

the following constant to be added :

INOUT = 'TERMINAL'; (\* filename for the terminal \*)

#### - At the variable declarations

var

the following new variables to be added :

TERM-IN, TERM-OUT : text ; (\* terminal input and  
output files \*)

- General correction

All the read and write statements must be modified  
as follows :

for read

reset(TERM-IN, INOUT);

read(TERM-IN, followed by the already existed  
parameters);

for write

rewrite(TERM-OUT, INOUT);

write(TERM-OUT, followed by the already existed  
parameters);

for eoln (input) or eoln without parameter

eoln(TERM-IN);

- The code of the procedure 'valuechar' will be  
substituted by the following code :

procedure valuechar (ch : cher; var k : integer );

begin (\* 1 \*)

k := ord (ch);

if (k >= 193) and (k <= 198) then

k := k - 193 (\* character A..F\*)

else if (k >= 240) and (k <= 249) then

k := k - 240; (\* digit 0..9\*)

```

end; (* 1 *)

- In the procedure 'setmem' :
  The statements :
    if temp < 10 then
      temp := temp + 48
    else
      temp := temp + 55;
  to be substituted by the following ones:
    if temp < 10 then
      temp := temp + 240 (* digit 0..9 *)
    else
      temp := temp + 183; (* character A..F *)

- In the procedure 'message' :
  The statements :
    if not eoln then
      read(cn)
    else begin
      ch := chr(13);
      readln;
    end;
  to be substituted by the statements :
    reset (TERM-IN, INOUT);
    if not eoln (TERM-IN) then
      read (TERM-IN, cn)
    else begin

```

```

      ch := chr(37);
      readln(TERM-IV);

```

```

    end ;

```

- In the procedure 'outputree' :

The statements :

```

      if length = 1 then
        write( chr( val1 mod 128 ));
        write ( chr( val2 mod 128 ));

```

to be modified as follows:

```

      rewrite(TERM-OUT , INOUT);
      if length = 1 then
        write(TERM-OUT,chr(val1));
        write(TERM-OUT,chr(val2));

```

- In the procedure 'INPUTOUTPUT' :

The statement

```

      write (chr(tempo) mod 128)

```

to be modified as follows :

```

      rewrite(TERM-OUT,INOUT);
      write( chr(tempo));

```

- In the procedure 'decode' :

The statement :

```

      opcode := (ord (line [j]) - 55) * hexv

```

to be modified :

```

      opcode := (ord (line [j]) -183) * hexv;

```

- In the procedure 'displaymix' :

The statement :

if (num > 31 ) and (num < 126) then

write( chr(num))

to be modified as follows :

if ( num > 64) then

write(TERM-OUT, chr(num));



## APPENDIX H

### USER'S INSTRUCTIONS

The user in order to run a program on the Z8000 simulator he needs to follow the following steps :

- compile the program 'simulator.p' using the command  
'pi simulator.p < cr > '
- transfer the obj code to a separate file using the command 'mv obj ex < cr > '
- compile the program assm.p (assembler) using the command 'pi assm.p < cr > '.
- transfer the obj code to a separate file using the command 'mv obj ase < cr > '.
- compile the program LOAD.p using the command 'pi LOAD.p < cr > '
- transfer the obj code to a separate file using the command 'mv obj load < cr > '.The above steps must be executed only once in order to be produced the object files.
- create a source file using either the equivalent hex code of the mnemonics or the mnemonics of the Z8000 ASM language as it is defined in appendix I.
- in the case of a mnemonics file type 'ase < cr > ' and wait a message from the assembler else type 'load

< cr >' and wait a message from the program LOAD.p.  
.And the two messages ask the user to enter the  
source filename.

- in the case of mnemonics source file the assembler  
produces a non readable object file in hex code and  
also it produces a listing file that contains the  
occurred errors during the assembling time , the  
generated code , the allocated space for variables and  
arrays and the original source file.The listing  
filename is equal to < sourcefilename.o >.

- the produced object file from the program LOAD.p is  
not readable hex code.

- the objects files that are produced from the  
assembler and the program LOAD.p are the default input  
files for the simulator

- type ' ex < cr >' and wait for the prompt ' < '.

- using the command 'L' load the program.

- set the RFC and the RPC registers using the  
commands 'R RFC <cr>' and 'R RPC < cr >'.The assembler  
provides the value of the RPC register with a message  
.

- execute the program using the command 'G' or 'Nn'

- reexecute the program clearing the registers using  
the command 'R R0 < cr >' and the memory that is  
affected by the execution with the command 'F address

address 0000 < or >'.  
- terminate the simulator using the command 'Q < or  
>' when the simulator is on the MONITOR program.

## APPENDIX I

### Z8002/ASM ASSEMBLER

The implemented in PDP 11/50 assembler using the PASCAL language is one pass assembler and it has the following basic characteristics:

- it permits the generation of object code up to the size of 2K.
- it supports constants, variables, labels, ORG statements and arrays. The number of permitted constants are 20, variables 30 and labels 30.

The above constraints have been imposed in order that the assembler uses real memory for the object code and not a file and so his speed of execution to be reasonable and second due to the limitations of the PDP 11/50 of run time object code up to 64K. The intention is to use the assembler to run on a main frame and so all the above restrictions will be lifted by changing only the values in the constants declarations and not affecting the structure of the program.

The input file is Z8002 assembly language in mnemonics and the produced object file is hexadecimal file which is the default input file for the implemented simulator. The supported language is described with BNF grammar in the following text.

The assembler also produces and a listing file where they are listed the generated code, the occurred errors during the assembling time and the source file.

The program it was tested by checking each one instruction  
all the combinations of the addressing modes .  
Two demo programs are provided after the listing of the  
program code.

BNF grammar of the supported language

module = declarations #

ENTRY

statements#

END

declarations = constants

= variables

constants = CONSTA

constant-definition #

constant-definition = constant-identifier

':=' constant

constant = number

= character-sequence

number = decimal-constant

= hex-constant

decimal-constant = digit + (till 2 to the power of 31

for all the kinds of values )

hex-constant = '%' hex-digit +

character-sequence = ' string-text + '

string-text = any -character, except quote

( the quote under quotes is defined with  
two quotes )

variable-identifier

lable-identifier

constant-identifier = letter ( letter | digit | '-' )  
( up to the maximum of 6 )

variables = VARIA

variable-initial-declaration \*

label-declaration \*

ORG-declaration \*

variable-initial-declaration = variable-identifier  
type ':' initial-value

type = BYTE

= WORD

= LONG

= ARRAY

ARRAY = [ decimal-constant BYTEE | WORD | LONG ]

initial-value = constant

( for the case of type BYTE,WORD,LONG )

initial-value = [ constant,constant ]

(for the case of array)(up to the number of the array  
dimension)

= [ constant . ]

( all the rest array components are initialized  
with the last value )

label-declaration = label-identifier LABEL



```

ORG-declarations = ORG constant
statement = label-identifier ':'
            = label-identifier ':' operation
            = operation
            = ORG-declaration
operation = Z6Z0Z-instruction
Z6Z0Z-instruction = opcode ( operands ) *
( number of operands depending on each
                    opcode )
operand = register
         = indirect-register
         = immediate
         = direct-address
         = indirect-address
         = relative-address
         = condition-code
         = flags
         = int
register = single-register
         = double-register
         = low-byte-register
         = high-byte-register
         = quad-register
         = special-register
single-register = R0 | R1 | R2 .... | R15
double-register = RR0 | RR2 .... | RR14

```

```

low-byte-register = RL0 | RL1 .... | RL7
high-byte-register = RH0 | RH1 .... | RH7
quad-register = RQ2 | RQ4 | RQ8 | RQ12
special-register = FLAGS | FCW | REFRES | PSAPSE |
                  PSAPOFF | NSPSEG | NSPOFF
indirect-register = 'Q' single-register
immediate = '#' constant
            = '#' variable-identifier
            = '#' constant-identifier
direct-address = constant
                = label-identifier
                = variable-identifier
                = constant-identifier
indexed-address = direct-address ( single-register )
relative-address = direct-address
                  = 'S' + constant-identifier | variable-identifier
                  | constant
condition-code = LT | LE | UL | OV | MI | EQ | C |
                TRUE | GE | GT | UGT | NOV | PL |
                NE | UGE |
flags = C | S | Z | P | V
int = VI | NVI
delimiter = space | tab | ','
comment = '!' any character
letter = 'A' | .... | 'Z' |
        'a' | .... | 'z' |

```

digit = '2' | .... | '9' |

hex digit = '0' | .... | '9' | 'A' | .... | 'F' |

'a' | .... | 'f' |

Z8000-instruction = | ADC | ADCB | ADD | ADDB | ADDL |  
| AND | ANDB | BIT | BITB | CALL | |
| CALR | CLR | CLRB | COM | COMB |  
| COMFLG | CP | CPB | CPI | CPD |  
| CPDB | CPDR | CPDRB | CPI | CPIB |  
| CPIX | CPIXB | CPSD | CPSDB | CPSDR |  
| CPSDRB | CPSI | CPSIB | CPSIR |  
| CPSIRB | DAB | DEJNZ | DEC | DECB |  
| DI | DIV | DIVL | DJNZ | EI | EX |  
| EXB | EXTS | EXTSB | EXTSL | FALT |  
| IN | INB | INC | INCB | IND | INDB |  
| INDR | INDRB | INI | INIB | INIR |  
| INIRB | IRET | JP | JR | LD | LDA |  
| LDAR | LDB | LDCTL | LDCTLB |  
| LDD | LDEB | LDDR | LDRB | LDI |  
| LDIB | LDIR | LDIRB | LDK | LDL |  
| LDM | LEPS | LDR | LDRB | LDRL |  
| MULT | MULTL | NEG | NEGB | NOP | OR |  
| ORB | OTDR | OTDRB | OTIR | OTIRB |  
| OUT | OUTB | OUTD | OUTDB | OUTI |  
| OUTIB | POP | POPL | PUSH | PUSHL |  
| RES | RESB | RESFLG | RET | RI |  
| RLB | RLC | RLDB | RLDB | RR | RRB |

```

| RRC | RRCB | RRDB | SBC | SBCE | SC
SDLL	SET	SETB	SETFLG	STORE	
STOREB	STOREL	SLA	SLAB	SLAL	
SLL	SLLB	SLLL	SRA	SRAB	SRAL
SRL	SRLB	SLLL	SUB	SUBB	SUBL
TCC	TCCB	TEST	TESTB	TESTL	
TRDB	TRDRB	TRTIB	TRTIRB		
TRTDB	TRTIB	TRTIRB	TSET		
TSETB	XOR	XORB			

```

maximum permitted values = BYTE = 255 | 1 char string |

2 nex char

= WORD = 65535 | 2 char string |

4 nex char

= LONG = maxint | 4 char string |

8 nex char

The difference between the store and load instructions in the Z8000 instruction set is denoted by the type of the operands. In the current implementation they are provided different mnemonics for the store instructions as follows : STOREB - STORE - STOREL

```
( **** PROGRAM      ASSEMBLER ASM / Z8000 *****)
```

```

label
100:
(* is used to terminate the program in the case that
an end of file is encountered in the input file
without to be proceeded by an end statement in the
code*)

```

```

61
62
63
64
65 const
66     filename = 17; (* maximum filename *)
67     maxline = 80; (* maximum length of the input line *)
68     margin = ' ';
69     maxaddress = 2000; (* maximum available memory of the assembler *)
70     maxcons = 20; (* maximum number of constants *)
71     maxvar = 29; (* maximum number of variables *)
72     maxlabel = 60; (* maximum number of labels = b0 - maxvar *)
73     stalabel = 30; (* the first label *)
74     maxint = 2147483647; (* maximum integer in the UNIX system *)
75     hexv = 16;
76     maxreg = 43; (* the number of the different names of registers *)
77     maxop = 168; (* the number of different names of opcodes *)
78     (* the following constants are used to change the base
79     value of the opcode depending on the type of the
80     addressing mode *)
81     IM = 0;
82     IR = 0;
83     R = 32768;
84     DA = 16384;
85     X = 16384;
86     maxarray = 100; (* maximum size of the arrays *)
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101

```



```

125 type
126 permit = (req, im, ir, da, xi, ba, bx,
127 cc, ra, control, nil);
128 (* is the set of the permitted types of operands *)
129 current = set of permit;
130 (* is used to produce the subsets of the permitted types
131 of operands depending on the opcode *)
132 one = packed array [0..5] of char;
133 alfa = packed array [1..mfilename] of char;
134 bravo =
135 record
136   name: one;
137   syntax: integer;
138   base: integer
139 end;
140 (* is used to create the table of the opcodes .the table
141 of the opcodes has the name of each opcode, the number
142 of operands and the base value of the opcode that cor-
143 responds to the ir address *)
144 charlie = packed array [0..maxaddress] of packed array [0..1] of char;
145 (* is the array that saves the produced object code *)
146 delta =
147 record
148   name: one;
149   initial, final: integer;
150   size: integer
151 end;
152 (* the record delta is used to create the variable and the
153 label table and it saves the name of each variable, or
154 label , the initial and final address for the varia-
155 bles and the size of the variable. the size of the varia-
156 ble is used to check overflow of the bounds of the array.
157 In the case of the labels it saves only his initial ad-
158 dress *)
159 echo = array [startlabel..maxlabel, 0..20] of integer;
160
161
162
163
164
165
166
167

```

```

181
182
183
184
185 (* this record is used to save the address that a label
186    is used without to have been met already and so the
187    assembler makes only one pass *)
188    foxtrot =
189    record
190        name: one;
191        value: real;
192        size: integer
193    end;
194 (*this record is used to save the values of the
195    declared constants *)
196    golf =
197    record
198        name: one;
199        value: integer;
200        lenght: integer
201    end;
202 (* this record is used to create the registers table
203    and contains the name of each register , his value
204    and his length (byte , word , long ) *)
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221

```

```

245 var
246
247 source, error, object: text;
248 (* source is the input file,error is the error file,
249 object is the object code file that is the default
250 input file of the simulator*)
251
252 init: file of bravo;
253
254 init1: file of golf;
255 (* the init and the init1 files are the files that are
256 produced by the program initialize.o which creates
257 the tables for the opcodes and the registers *)
258
259
260 infile, errorfile, outfile: alfa;(* filenames for the above *)
261 (* files *)
262
263 opcode: array [1..maxop] of bravo;
264 (* this array contains all the names of the opcodes*)
265
266 fast : array [1..185] of integer;
267 (* this array contains the link list of the
268 opcodes by alphabetical order and the search
269 of the opcodes is restricted in a subset depen
270 ding on the first character *)
271
272 op: integer; (* index in the above array *)
273
274 linkheader : array [1'A'..'X'] of integer;
275 (*this array has the headers of the link lists
276 of the opcodes by alphabetical order *)
277
278 memory: charlie;(* memory array that saves the object code*)
279
280
281
282
283
284
285
286
287

```

```

301
302
303
304
305 variable: array [0..maxlabel] of delta;
306 (* records for the declared variables *)
307
308 constlist: array [0..20] of foxtrot;
309 (* array to save the values and the attributes of the constants *)
310
311 PC: integer; (* is the program counter *)
312
313 empty, card: packed array [1..81] of char;
314 (* card input buffer that saves a complete line *)
315 (* empty is empty buffer and is used to clear the
316    the input buffer after each use *)
317
318 ch: char;
319 (* is used to save the last character from the buffer *)
320
321
322 errorcounter: integer;
323 (* counts the number of the occurred errors *)
324
325 cdp: integer; (* character counter *)
326
327 ll: integer; (* line length *)
328
329 register: array [0..maxreg] of golf;
330 (* contains all the names of the registers and their
331    corresponding values *)
332
333 labelist: echo; (* array to store the addresses of the labels *)
334 (* that had been referred without to be encountered yet *)
335
336 token: one;
337 (* is used to find the tokens either constants
338    variables or oncodes or registers names or other operands *)
339
340
341
342

```

```

365 end2: set of char;
366 (* is used to create set of delimiters characters *)
367
368 last: char;
369 (* is the inserted last character in the
370 input buffer for checking purposes *)
371
372
373 constindex, varindex, labelindex: integer;
374 (* indexes in the tables of the constants, variables and
375 labels *)
376
377 kindtypes: array [0..51] of one;
378 (* is array of the permitted types of variables *)
379 type1, type3, type4, type5, type6, type7, type8, type9: current;
380 (* subsets of the permitted types of operands depending
381 on the type of the opcode *)
382
383 finished: boolean;
384 (* denotes normal termination of the program *)
385
386 controlreg: array [0..6] of one;
387 (* array of the names of all the control registers *)
388
389 condition: array [0..15] of one;
390 (* array of the names of all the condition codes *)
391
392 LAST, START: real;
393 (* LAST denotes the address of the last encountered
394 ORG statement and START denotes the starting address
395 of the RPL register in the monitor program *)
396
397 theend: boolean;
398 (* this variable is used to denote if the procedure dechex
399 will write to the errorfile or the screen *)
400
401
402
403
404
405
406
407

```

(procedure trace;

this procedure is used only for debugging  
purposes so is under comments ,it can be  
used to print the values of all the con-  
stants,variables,labels and the contents  
of the memory

```

var
  op, i: integer;

begin
  writeln('CONSTANT');
  for op := 0 to constindex do
    writeln('name ', constlist[op].name, ' value ',
      constlist[op].name:16:2, ' size ',constlist[op].size);
  writeln('VARIABLES');
  for op := 0 to varindex do
    with variable[op] do
      writeln(' name ', name, ' initial - final ',
        initial : 6,final: 6,' size ',size:4);
    for op := 40 to labelindex do
      with variable[op] do
        writeln(' name ', name, ' initial-final ',
          initial: 6, final: 6, ' size ', size: 4);
      for op := 0 to pc -1 do begin
        if op mod 2 = 0 then
          writeln;
        for i := 0 to 1 do begin
          write(memory[op][i])
        end
      end
    end;
  end;
  (* trace *)

```



```

605 procedure dechex(n: integer);
606
607
608 (*The procedure is used to trasform a decimal number
609 to equivalent hex and then to print the result to
610 the screen.For this reason uses an array of 4 chara-
611 cters which is initialized to spaces.
612 - hexval is the used array.
613 - temp is the result of the mod operation of the
614 remaining value divided by 16.Range 0..15.
615 - num is the quotient of the value divided by 16.
616 Range 0..maxinteq.
617
618 *)
619 var
620 j, temp, num: integer;
621 hexval: packed array [0..3] of char;
622
623 begin (*2*)
624 j := 3;
625 hexval := ' ';(*fills the array with spaces*)
626 num := n; (* denotes the number to be trasformed from*)
627 (*decimal to hex value*)
628 while j >= 0 do begin (*2*)
629 temp := num mod 16;
630 if temp < 10 then
631 temp := temp + 48 (* digit 0..9 *)
632 else
633 temp := temp + 55;(* character A..F *)
634 hexval[j] := chr(temp);(* fill the array *)
635 num := num div 16;
636 j := j - 1
637 end; (*2*)
638 if not thenend then
639 write( error, ' ':4,hexval, ' ')(* write the word to the error *)

```



```
661
662
663
664
665      (* file *)
666      else
667          writeln (' !:?,hexval);
668      end; (* ! dechex *)
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
```

procedure error\_messages is  
 (\* this procedure prints the error messages to the  
 errorfile and also indicates with arrow where the  
 error had occurred. The number of the error is passed  
 with the parameter n and the current position of the  
 character where the error has occurred is known from  
 the value of the global variable cdp. It also increases  
 the value of the variable errorcounter in  
 each error\*)

```

begin
  writeln(error, ' ':20, ' ': cdp - 1, '!', n: 3);
  errorcounter := errorcounter + 1;
  write (error, ' ':20);
  case n of
    1:
      writeln(error, 'Line length > 80 characters');
    2:
      writeln(error, 'First character must be 'A'..'Z');
    3:
      writeln(error, 'More than 20 constants are used');
    4:
      writeln(error, ' := expected');
    5:
      writeln(error, 'Identifier expected and no reserved word');
    6:
      writeln(error, 'Character or hex or digit expected');
    7:
      writeln(error, chr(39), ' expected');
    8:
      writeln(error, 'Hex character expected');
    9:
      writeln(error, 'End statement expected');
    10:
      writeln(error, 'Decimal digit expected');
  end case;
end;
```

```

781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
11:
writeln(error, 'Constant has been defined already');
12:
writeln(error, 'Variable has been defined already');
13:
writeln(error, 'Variable has been defined as constant');
14:
writeln(error, 'More than 30 labels are used');
15:
writeln(error, 'More than 30 variables are used');
16:
writeln(error, 'No correct declaration of array');
17:
writeln(error, 'Dimension of array > 100');
18:
writeln(error, 'Uppcode expected');
19:
writeln(error, 'No more memory');
20:
writeln(error, 'Greater value than type permits');
21:
writeln(error, 'Invalid character');
22:
writeln(error, 'Expected initial value for the variable');
23:
writeln(error, 'No correct operand ');
24:
writeln(error, ' : expected ');
25:
writeln(error, 'Uppcode and not label expected');
26:
writeln(error, 'Incompatible register (s) for the opcode');
27:
writeln(error, 'The label is used > 20 times forward');
28:

```

```

845 writeln(error, 'No correct operand (s)');
846 29:
847 writeln(error, 'Out of bounds of the array');
848 30:
849 writeln(error, '] expected');
850 31:
851 writeln(error, '); expected');
852 32:
853 writeln(error, 'Register name expected');
854 33:
855 writeln(error, ', expected');
856 34:
857 writeln(error, 'Second register incompatible ');
858 35:
859 writeln(error, 'No correct flag or the flag has already',
860 ' been used');
861 36:
862 writeln(error, 'The last used OK6 or PC had higher address');
863 37:
864 writeln(error, 'An array is of type byte or word or long');
865 38:
866 writeln(error, ' + expected ');
867 39:
868 writeln(error, 'Label has been used in previous',
869 ' statement ');
870 40:
871 writeln(error, 'Type declaration expected');
872
873 end
874 end; (* errormessage *)

```

```

901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941

```

```

    procedure kit;

    (* this procedure is called to retreat the character
       pointer in the input buffer when the program meets a
       reserved word as ENTRY and must change searching algorithm
       so it changes algorithm and starts again to search the
       input buffer *)

    begin
        cdp := 1;
        ch := cardfl;
        end; (* kit *)

```

```

965 procedure clear;
966
967 (* this procedure is called to clear the input buffer
968    after the completion of the analysis of the
969    current line. The analysis terminates in the
970    following conditions :
971    - error condition
972    - ! character has encountered
973    - last character of the input buffer
974      was met.
975    *)
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007

```

```

1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062

procedure load;
(* this procedure performs two functions :
   -loads a new line to the inout buffer from the
     source file.
   - returns a character to the calling procedure
     till all the characters of the buffer has been
     analyzed and then repeats the above step
The procedure loads a new line only if the variables
cdp character pointer and ll line length are equal
else increases the character pointer and returns the
next character *)

begin
  if cdp = ll then begin
    (* loads a new line *)
    if eof(source) then begin
      errormessage(9);
      goto 100
    (* if eof encounters then terminates the program,
       this means that the input source file does
       not provide an end statement and this situation
       is abnormal termination.It also prints an error-
       message to the errorfile *)
    end;
    write (error, ' :20 ');
    while not eofn(source) and (ll < maxline) do begin
      ll := ll + 1;
      read(source, ch);
      if ch in ['a'..'z'] then
        ch := chr(ord(ch) - 32);
      (* checks if the character is small and
         changes that to capital so by this method
         decreases the size of the tables for
         searching*)
      cardllll := ch;

```



```

1085 write(error, ch);
1086 (* it transfers to the errorfile the read
1087    character *)
1088 card[l] + 1] := last
1089 (* sets the sentinel character and so the
1090    searching procedures know the end of the
1091    input line *)
1092 end;
1093 if not eoln(source) and (l) = maxline) then begin
1094   repeat
1095     (* if the line is greater than 80 characters then
1096        continuous to transfer to the errorfile the
1097        rest of the line and it prints an errormessage*)
1098     read(source, ch);
1099     write(error, ch)
1100   until eoln(source);
1101   cdp := #0;
1102   errormessage(1);
1103   cdp := 0
1104   end;
1105   readln(source);
1106   writeLn(error);
1107   ll := ll + 1
1108 end;
1109 if cdp < 1) then begin
1110   (* in all the other cases it returns an
1111      character to the calling procedure *)
1112   cdp := cdp + 1;
1113   ch := card[cdp]
1114 end
1115 end; (* load *)

```

```

1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127

```

```

1141
1142
1143
1144
1145
1146
1147 procedure skiplines;
1148 (*this procedure skips empty lines
1149 so it calls the procedure load that
1150 loads new lines if the cdp = 11*)
1151
1152 begin
1153 while card = empty do begin
1154   cdp := 0;
1155   11 := 0;
1156   load
1157 end
1158 end; (* skiplines *)
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182

```

```

1205 procedure skipblanks(n: integer);
1206
1207 (* this procedure searches to find the first
1208    no blank character or to find the first ',' or
1209    to find the '!' as defined by the parameter 'n'.
1210    the procedure load sets as last character the chr(0)
1211    so the skipping terminates if the desired character
1212    is encountered or the last character of the input
1213    buffer is encountered. *)
1214
1215 begin
1216   case n of
1217     0:
1218       while ((ch = ' ') or (ch = chr(9))) and not (ch = last)
1219       do
1220         load;
1221       l:
1222       begin
1223         while (ch in [' ', chr(9)]) and not (ch = last)
1224         and not (ch = ',') do
1225           load;
1226         if ch <> ',' then
1227           errormessage(33)
1228           (* this situation is used to find the ',' that
1229              separates initial values of an array or the
1230              different operands. So if then ',' is not en-
1231              countered it prints an error message *)
1232         end;
1233       2:
1234       begin
1235         if not (ch in ['!', chr(0)]) then begin
1236           while (ch in [' ', chr(9)]) do
1237             load;
1238         if not (ch in ['!', chr(0)]) then
1239           errormessage(21)

```

```

1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302

(* after the last operand or the assignement
   values in the case of constants and varia-
   bles it is not permitted to exist comments
   without to be proceeded by the '!' symbol *)
end
end
end; (* skipblanks *)

```

```

1325 procedure skipcomments;
1326 (* the procedure skips the lines with only comments
1327 or blanks ones so it calls the procedure skipblanks
1328 and the procedure skiplines *)
1329 begin
1330 repeat
1331   skiplines;
1332   skipblanks(0);
1333   if ch = '!' then
1334     clear
1335   until ch <> '!';
1336 end; (* skipcomments *)
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367

```

```

1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422

procedure getassign(var correct: boolean);
(* the procedure scans the input file for the ':'=
assignment symbol .The assignment symbol is used
in the constants declarations and the variables
declarations ,so at first it skips the spaces and
if the assignment symbol is not encountered it
prints an error message and sets the parameter
correct to false *)

begin
  correct := false;
  skipblanks(0);
  if ch = ':' then begin
    load;
    if ch = '=' then
      correct := true;
  end;
  if not correct then
    errormessage(4)
  end; (* getassign *)

```

```

1445 procedure gettoken(n: integer; var token: one; var correct: boolean);
1446 (* this procedure is called to return the next token
1447    with the parameter token. So it calls repeatedly
1448    the procedure load to return the next character and
1449    terminates in the following conditions:
1450    - the first character is not in 'A'..'Z'
1451    - delimiter has been encountered
1452    - the specified length of the token
1453      has been met without to find deli-
1454        miter character.
1455    If an error occurs then it prints an error message and
1456    also it sets the parameter correct to false *)
1457
1458 var
1459   count: integer;
1460
1461 begin
1462   correct := true;
1463   token := ' ';
1464   (* it clears the value of the token to
1465      prevent the existence of old characters
1466      in the variable *)
1467   count := 0;
1468   repeat
1469     if (count = 0) and not (ch in ['A'..'Z']) then begin
1470       errormessage(2);
1471       correct := false;
1472     end;
1473     token[count] := ch;
1474     count := count + 1;
1475     load
1476   until (ch in end2) or (count > n) or not correct
1477   or (ch in [' ', '(', ')', ',', '!', ':', ';', '}', '{', '}', '|']);
1478   end; (* gettoken *)
1479
1480
1481
1482
1483
1484
1485
1486
1487

```



```

1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602

procedure give (var index : integer; fi : char );

(*This procedure depending on the first character of
the opcode returns the header of a link list so as
the search to be restricted in a subset of the op
code table *)

begin
    if fi in [ 'A','B','C','D','E','I','J','L',
               'M','N','O','P','Q','R','S','T','X' ] then
        index := linkheader (fi)
    else
        index := linkheader ['H'];
    end;

```

```
procedure check(var correct: boolean; token: one; number: integer);
```

```
(* This procedure is used by the variable and  
constant algorithms .In both cases it checks  
if the declared constant or variable or label  
is a reserved word as :
```

- oncode
- register name
- control register
- condition code

```
If in the reality it is an reserved word then  
it prints an error message and sets the parameter  
correct to false.
```

```
In the case of constants it checks also if the  
constant has been declared already.
```

```
In the case of variable or labels it checks if  
the label or variable has been declared already  
as constant or variable or label and then prints  
an error message and sets the parameter correct  
to false *)
```

```
(* The search is restricted in subsets of the opcodes  
depending on the first character of the token,  
also in the case of the registers tables it searches  
only if the token starts from K and in the conditions  
tables only if the token has no more than 4 characters*)
```

```
var
```

```
lma,m,l,i, k: integer;
```

```
begin
```

```
k := 0;
```

```
i := -1;
```

```
m := -1;
```

```
l := 0;
```

```
1625  
1626  
1627  
1628  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649  
1650  
1651  
1652  
1653  
1654  
1655  
1656  
1657  
1658  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667
```

```

1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1722

correct := true;
if token[0] = 'R' then begin (* check for register*)
  repeat
    i := i + 1;
  until (token = registerfil.name) or (i = maxreq);
  if token = registerfil.name then
    correct := false;
end; (* register*)
if correct then begin (* continues in the oocodes *)
  (* only if it was not found in the registers*)
  give (lima, token[0]);
  repeat
    k := fast (lima);
    lima := lima + 1;
  until (opcode[k].name = token) or (fast(lima) = -1);
  (* till to find the opcode or the end of the link
  list *)
  if (opcode[k].name = token) then
    correct := false;
end; (* oocodes*)
if correct and (token[4] = ' ') then (* continues in
the condition code table *)
  begin
    repeat
      l := l + 1;
    until (token = condition[l]) or (l = 15);
    if token = condition[l] then
      correct := false;
    end;
    if correct then begin
      (* continue in the control registers *)
      repeat
        m := m + 1;
      until (token = controlreg[m]) or (m = 6);
      if controlreg[m] = token then

```

```

1745 correct := false;
1746 end;
1747 if not correct then (* only if it is correct *)
1748   errormessage(5);
1749   if correct then
1750     case number of
1751       1:
1752         (* checks for the constants *)
1753         begin
1754           k := 0;
1755           while (token <> constlist[k].name) and (k < constindex - 1) do
1756             k := k + 1;
1757           if token = constlist[k].name then begin
1758             errormessage(11);
1759             (* if the constant has already been declared then
1760              it prints an errormessage *)
1761             correct := false
1762           end
1763         end;
1764       2:
1765         begin
1766           (* case of variables and labels *)
1767           k := 0;
1768           i := 0;
1769           while (token <> variable[k].name) and (k < varindex) do
1770             k := k + 1;
1771           if token = variable[k].name then begin
1772             errormessage(12);
1773             (* if the variable or the label has been used
1774              as variable then it prints an errormessage *)
1775             correct := false
1776           end;
1777           if correct then begin
1778             k := 40;
1779             while (token <> variable[k].name) and (k < labelindex) do

```

```
1801
1802
1803
1804
1805      k := k + 1;
1806      if token = variable[k].name then begin
1807          errormessage(12);
1808          (* if the variable or the label has been
1809             used as an label then it prints an error
1810             message *)
1811          correct := false
1812      end
1813      end;
1814      k := 0;
1815      while (token <> constlist[k].name) and (k < constindex) do
1816          k := k + 1;
1817      if token = constlist[k].name then begin
1818          errormessage(13);
1819          (* if the variable or the label has been used
1820             as a constant then prints an errormessage *)
1821          correct := false
1822      end
1823      end
1824      end; (* check *)
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834 >
1835
1836
1837
1838
1839
1840
1841
```

```

procedure decimal(var value:real; var correct:boolean);
size := integer;
(* The procedure is called to transform a decimal
   string of characters of the input buffer to his
   equivalent decimal value. The calling procedure
   must specify the size of the expected value as
   byte or word or long.
   The search terminates in the following conditions:
   - no decimal digit was encountered
   - space or delimiter is encountered
   - the value of the number is greater than
     then the specified size by the calling pro-
     cedure.
   If an error condition occurs then sets the para-
   meter correct to false and also it prints an error-
   message *)
var
max: integer;
begin
  if size = 2 then
    max := 255 (* byte *)
  else if size = 4 then
    max := 65535 (* word *)
  else if size = 8 then
    max := maxint; (* long word *)
  value := 0;
  correct := true;
  repeat
    if not (ch in end2) then begin
      if ch in '0'..'9' then
        value := value * 10 + ord(ch) - 48
      else begin
        errormessage(10);

```

```

1921
1922
1923
1924
1925
1926      (* no digit encountered *)
1927      correct := false
1928
1929      end
1930      end;
1931      load
1932
1933      until (value > max) or (ch in end?) or not (ch in ['0'..'9'])
1934      or (ch = ',') or (ch = ' ');
1935      if value > max then begin
1936          (* greater value than the specified has been calculated *)
1937          errormessage(20);
1938          correct := false
1939      end
1940      end; (* decimal *)
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961

```



```

1985 procedure string(var number: real; var correct: boolean;
1986 size : integer);
1987
1988 (* This procedure is used to transform a string of
1989 characters in their equivalent decimal values.
1990 So each character increases the already calculated
1991 value by 256 * the value of the current
1992 character. The calling procedure is responsible
1993 to define the length of the string. The transformation
1994 terminates in the following conditions:
1995 - the size of the string is greater from
1996 the specified by the parameter size
1997 - last character of the input buffer has
1998 been encountered .
1999 - left '...' has been encountered .
2000
2001 If an error condition occurs then it prints
2002 an error message and also it sets the parameter
2003 correct to false *)
2004
2005 var
2006 i: integer;
2007 max: integer;
2008 quote : boolean;
2009
2010 begin
2011 max := size div 2;
2012 quote := false;
2013 (* case size of
2014 -2 then max = 1 byte
2015 -4 then max = 2 word
2016 -8 then long = 4 long*)
2017
2018 i := 0;
2019 number := 0;
2020 correct := true;
2021 repeat
2022
2023
2024
2025
2026
2027

```

```

2041
2042
2043
2044
2045
2046 load;
2047 if (ch <> chr(39)) or (card (cddp +1) = chr (39)) then
2048 (* it checks if the next character is also
2049 '...' and it permits to be included in the
2050 the string a '...' *)
2051 number := number * 256 + ord(ch);
2052 if (ch = chr(39)) and (card (cddp +1) = chr (39)) then
2053 begin
2054 quote := true;
2055 load;
2056 (* skips the next '...' character *)
2057 end else
2058 quote := false;
2059 i := i + 1
2060 until (ch = last) or (i > max)
2061 or (( ch = chr (39) ) and not quote );
2062 if ch <> chr(39) then begin
2063 if i <= max then
2064 errormessage(7);
2065 (* always the string must terminates with left '...' *)
2066 if i > max then
2067 errormessage(20);
2068 (* greater value was encountered *)
2069 correct := false
2070 end else
2071 load
2072 end; (* string *)
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082

```

```

2104 procedure hexvalue(var number: real; var correct: boolean;
2105 size: integer);
2106
2107

```

```

2108 (* This procedure is used to transform hex characters
2109 to their equivalent decimal value. The calling pro-
2110 cedure must specify the size of the hex value. The
2111 transformation terminates in the following condi-
2112 tions :

```

- a character that is not hex is encountered
- a delimiter is encountered
- the calculated value is greater than the  
the specified by the caller.

```

2117 In the case of error it prints an error message and
2118 it sets the parameter correct to false. *)
2119

```

```

2120 var
2121 i: integer;

```

```

2122 begin

```

```

2123 number := 0;

```

```

2124 i := 0;

```

```

2125 correct := true;

```

```

2126 repeat

```

```

2127   load;

```

```

2128   if not ( (ch in end2) or (ch = ',') ) then begin

```

```

2129     if ch in ['0'..'9', 'A'..'F'] then begin

```

```

2130       if ch in ['0'..'9'] then

```

```

2131         number := number * hexv + ord(ch) - 48

```

```

2132       else

```

```

2133         number := number * hexv + ord(ch) - 55;

```

```

2134         (* then hex character A..F *)

```

```

2135         i := i + 1

```

```

2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202

    end else begin
        correct := false;
        (* no hex character *)
        errormessage(8)
    end
end else begin
    if i = 0 then begin
        errormessage(8);
        (* means that does not exist any hex character
        for conversion *)
        correct := false
    end
end
until (i > size) or (ch in end2) or not (ch in ['0'..'9', 'A'..'F'])
or not correct or (ch = ',') or (ch = '(');
(* case size of
    -2 then two hex characters so byte
    -4 then four hex characters so word
    -8 then eight hex characters so long*)
if (i >= size) and not ( (ch in end2) or (ch = ',')
or (ch = '(')) then begin
    errormessage(20);
    (* means that exist and other characters for conversion *)
    correct := false
end
end; (* hexvalue *)

```

```

2225 procedure evalconst(var value: real; var correct: boolean;
2226 size: integer);
2227 (* This procedure is used to find the type of the
2228 the value of the constant and then calls the corres-
2229 ponding evaluation procedure as hexvalue, decimal or
2230 string. The constants are evaluated in 32 bits arith-
2231 metic so the parameter size has value = 8. In the
2232 case that the value of the constant is not correct
2233 it prints an error message *)
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
begin
    correct := true;
    skipblanks(0);
    if ch = chr(39) then
        string(value, correct, size)
    else if ch = '%' then
        hexvalue(value, correct, size)
    else if ch in ['0'..'9'] then
        decimal(value, correct, size)
    else
        errormessage(6)
    end; (* evalconst *)

```

```

2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322

procedure retreat(n: integer);
(* this procedure is used to retreat the indexes
   of the variable and constant tables in the case
   that the variable or the constant is not correct*)
begin
case n of
0:
constindex := constindex - 1;
1:
varindex := varindex - 1;
end
end; (* retreat *)

```

```

procedure gettype(var kind: integer; var correct: boolean);

(* This procedure is used by the variable scanner to search
   and to find the type of the variable :
   byte - word - long - array - label
   The type of the variable is equal to the double value
   of the index of the array that contains the names
   of the types .In the case that the type is not correct
   it sets the parameter correct to false*)

var
  index: integer;
begin
  correct := true;
  skipblanks(0);
  gettoken(5, token, correct);
  (* calls the procedure gettoken to return the next token *)
  if correct then begin
    (* if the token is correct then searches for the kind of
       the variable *)
    index := -1;
    repeat
      index := index + 1
    until (kindtypes[index] = token) or (index = 5);
    if kindtypes[index] = token then
      kind := index * 2
    (* if the token is ewqual to one of the stored values in
       in the array it returns the double of the value
       of the index *)
    else
      correct := false
    end
  end;
end;
  
```



```

2461 procedure setmem(index: integer; value: real; length: integer);
2462
2463 (*this procedure is used after the evaluation of and variable
2464 or the encoding of an opcode to set the virtual memory to
2465 the corresponding hex value. The procedure at first it cal-
2466 culates the offset and then it checks for overflow or under-
2467 flow. In the case of underflow the error is originated from
2468 the assembler and not from the user. The procedure in order to
2469 change the decimal values to the corresponding hex it uses
2470 her mod function because the values may be greater than the
2471 maximum integer of the UNIX system *)
2472
2473 var
2474   val: real;
2475   ind, hyt, temp, j: integer;
2476   hexval: packed array [1..8] of char;
2477   (* this array is used to print the code
2478      to the errorfile in the correct order *)
2479   hpoint: integer; (* is the index of the array*)
2480   alldone: boolean;
2481
2482 begin
2483   alldone := false;
2484   if length = 8 then
2485     hpoint := 8
2486   else if length = 4 then
2487     hpoint := 4
2488   else
2489     hpoint := 2;
2490   hexval := '0000'; (* clears the array *)
2491   (* alldone is used to terminates the procedure in the
2492      case of overflow or underflow *)
2493   val := value;
2494   dechex (index);
2495   (* it prints the PC at the error file *)

```

```

2525 if length > 2 then
2526   ind := index + length div 2 - 1
2527   (* calculates the offset *)
2528 else
2529   ind := index;
2530   (*if ind - length div 2 < -1 then begin
2531     checks for underflow
2532     errormessage( );
2533     alldone := true
2534   end; *)
2535   if ind > maxaddress then begin
2536     (* checks for overflow *)
2537     alldone := true;
2538     errormessage(19)
2539   end;
2540   byt := 1;
2541   (* always starts from the higher byte and the
2542     higher address *)
2543   if not alldone then begin
2544     (* if not error condition then permits the change *)
2545     for j := length downto 1 do begin
2546       temp := trunc(val / hexv - trunc(val / nexv)) * hexv);
2547       if temp < 10 then
2548         temp := temp + 48
2549       (* means that it is digit 0..9 *)
2550     else
2551       temp := temp + 55;
2552     (* else it is hex character A..F *)
2553     memory[ind]bytl := chr(temp);
2554     hexval[hpoint] := chr(temp);
2555     (* sets the array *)
2556     hpoint := hpoint - 1;
2557     (* it transfers to the error file the generated code *)
2558     val := trunc(val / hexv);
2559     byt := byt - 1;
2560
2561
2562
2563
2564
2565
2566
2567

```

```

2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622

    if byt < 0 then begin
        ind := ind - 1;
        (* means that it has set two bytes and so it must
           decreases the memory index *)
        byt := 1
        (* resets the byte index to the high value *)
    end
end;
writeln (error,hexval);
end;
end; (* setmem *)

```

```

2645 procedure getdimension(var size, kind: integer;
2646   var correct : boolean );
2647
2648 (* This procedure is used to find the dimension and the type of
2649   an array .The size and the type of the array must be enclosed
2650   in [ ] so the procedure it also checks for the existence of the
2651   delimiters.If the size of the array is greater than 100 then
2652   it prints an error message and sets the size of the array at
2653   30.After the calculation of the size it calls the procedure
2654   gettype to return the type of the array.If the type of the
2655   array is array or label it prints an error message and sets
2656   the parameter correct to false*)
2657
2658 var
2659   value: real;
2660
2661 begin
2662   correct := true;
2663   skipblanks(0);
2664   if ch = '[' then begin
2665     (* checks for left [ *)
2666     load;
2667     decimal(value, correct, 2);
2668     (* calls the procedure decimal to calculate the
2669       dimension of the array *)
2670     if correct then begin
2671       if value > maxarray then begin
2672         errormessage(17);
2673         value := maxarray
2674       end;
2675       size := trunc(value);
2676       gettype(kind, correct);
2677       if correct then begin
2678         if (kind = 0) or (kind = 10 ) then
2679           begin
2680
2681
2682
2683
2684
2685
2686
2687

```

```

2701
2702
2703
2704
2705      errormessage ( 37 );
2706      correct := false;
2707      end;
2708      (* means that the array is not possible to
2709       * of type array or label *)
2710      skipblanks(0);
2711      if ch <> ']' then begin
2712        correct := false;
2713        (* ) expected *)
2714        errormessage (30);
2715        end;
2716      end;
2717      end
2718
2719      end
2720      end; (* getdimension *)
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741

```

```

2764 procedure calconstant(var alldone: boolean);
2765
2766 (* This procedure performs the following operations :
2767
2768 - skips empty lines and lines with only comments
2769 - in the first no empty line it finds the first token.
2770 - checks if the token equals to 'VARIA' or 'ENIRY'.
2771 In either case it terminates the procedure because
2772 the search must continue with the variable analyzer
2773 or the opcode analyzer.
2774 - if the token is not equal to one of the above values
2775 then it is constant. So it increases the constant
2776 index in the constant table and checks if they have
2777 used more than 20 constants.
2778 - If it is room in the constant table for the new constant
2779 it checks if the constant is a reserved word or an already
2780 declared constant.
2781 - if the results of the search are correct then it
2782 searches for the ':' symbol and then for the value
2783 of the constant calling the procedure evalconst. Finally
2784 it sets the entry in the constant table with the value
2785 of the constant and the size of the constant. All the
2786 constants are calculated in 32 bit arithmetic.
2787 - If an error occurs it prints an error message and
2788 decreases the constant index in the constant table.
2789 - The variable alldone is set to true only if the token
2790 equals to 'VARIA' or 'ENIRY' denoting that there are
2791 not more constants for analysis.
2792 *)
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807

```

var







```

2884 end
2885 end (*6*); (*4*)
2886 if not correct then
2887   retreat(0)
2888   (* reduces the index of the constants *)
2889 end else begin (* 3 5a*)
2890   errormessaqe(3);
2891   retreat(0)
2892 end; (*3a*)
2893 clear
2894 (* clear the input buffer *)
2895 end else begin (* 2 2a*)
2896   if correct then
2897     alldone := true
2898     (* 'VARIA' or 'ENTRY' is encountered *)
2899   else
2900     clear
2901   end (* 2 *)
2902 end; (* 1 *) (* calconstant *)
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927

```

```

2941
2942
2943
2944
2945 procedure constantanalyze;
2946
2947 (* This procedure performs the analysis of the constants
2948    calling repeatedly the procedure calconst.
2949    The procedure terminates if the procedure calconst re-
2950    turns the parameter alldone set to true.
2951    The procedure in order to start the analysis of the con-
2952    stants it must encountered the reserved word CONSTIA so
2953    it skips all the empty lines and the lines with only
2954    comments with the procedure skipcomments. In the case
2955    that the first no empty line has a token different than
2956    the reserved word CONSTIA then retreats the character
2957    pointer in the input buffer and terminates the search.
2958    *)
2959
2960
2961 var
2962 correct, alldone: boolean;
2963
2964
2965 begin
2966 alldone := false;
2967 skipcomments;
2968 (* skips empty lines and lines with comments *)
2969 gettoken(5, token, correct);
2970 if token = 'CONSTIA' then begin
2971   clear;
2972   repeat
2973     skipcomments;
2974     calconstant(alldone)
2975   until alldone
2976 end else
2977   kit;
2978 (* retreat the character pointer in the input buffer *)
2979 if alldone then
2980
2981
2982

```

3004  
3005  
3006  
3007  
3008  
3009  
3010  
3011  
3012  
3013  
3014  
3015  
3016  
3017  
3018  
3019  
3020  
3021  
3022  
3023  
3024  
3025  
3026  
3027  
3028  
3029  
3030  
3031  
3032  
3033  
3034  
3035  
3036  
3037  
3038  
3039  
3040  
3041  
3042  
3043  
3044  
3045  
3046  
3047

kit  
end; (\* constanalyze \*)

```

3061
3062
3063
3064
3065
3066
3067
3068
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3360
3361
3362
3363
3364
3365
3366
3367
3368
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3379
3380
3381
3382
3383
3384
3385
3386
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
3400
3401
3402
3403
3404
3405
3406
3407
3408
3409
3410
3411
3412
3413
3414
3415
3416
3417
3418
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
3450
3451
3452
3453
3454
3455
3456
3457
3458
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3479
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
3500
3501
3502
3503
3504
3505
3506
3507
3508
3509
3510
3511
3512
3513
3514
3515
3516
3517
3518
3519
3520
3521
3522
3523
3524
3525
3526
3527
3528
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
3550
3551
3552
3553
3554
3555
3556
3557
3558
3559
3560
3561
3562
3563
3564
3565
3566
3567
3568
3569
3570
3571
3572
3573
3574
3575
3576
3577
3578
3579
3580
3581
3582
3583
3584
3585
3586
3587
3588
3589
3590
3591
3592
3593
3594
3595
3596
3597
3598
3599
3600
3601
3602
3603
3604
3605
3606
3607
3608
3609
3610
3611
3612
3613
3614
3615
3616
3617
3618
3619
3620
3621
3622
3623
3624
3625
3626
3627
3628
3629
3630
3631
3632
3633
3634
3635
3636
3637
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
3650
3651
3652
3653
3654
3655
3656
3657
3658
3659
3660
3661
3662
3663
3664
3665
3666
3667
3668
3669
3670
3671
3672
3673
3674
3675
3676
3677
3678
3679
3680
3681
3682
3683
3684
3685
3686
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
3700
3701
3702
3703
3704
3705
3706
3707
3708
3709
3710
3711
3712
3713
3714
3715
3716
3717
3718
3719
3720
3721
3722
3723
3724
3725
3726
3727
3728
3729
3730
3731
3732
3733
3734
3735
3736
3737
3738
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749
3750
3751
3752
3753
3754
3755
3756
3757
3758
3759
3760
3761
3762
3763
3764
3765
3766
3767
3768
3769
3770
3771
3772
3773
3774
3775
3776
3777
3778
3779
3780
3781
3782
3783
3784
3785
3786
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799
3800
3801
3802
3803
3804
3805
3806
3807
3808
3809
3810
3811
3812
3813
3814
3815
3816
3817
3818
3819
3820
3821
3822
3823
3824
3825
3826
3827
3828
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
3850
3851
3852
3853
3854
3855
3856
3857
3858
3859
3860
3861
3862
3863
3864
3865
3866
3867
3868
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
3900
3901
3902
3903
3904
3905
3906
3907
3908
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
3950
3951
3952
3953
3954
3955
3956
3957
3958
3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
4000
4001
4002
4003
4004
4005
4006
4007
4008
4009
4010
4011
4012
4013
4014
4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
4050
4051
4052
4053
4054
4055
4056
4057
4058
4059
4060
4061
4062
4063
4064
4065
4066
4067
4068
4069
4070
4071
4072
4073
4074
4075
4076
4077
4078
4079
4080
4081
4082
4083
4084
4085
4086
4087
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
4100
4101
4102
4103
4104
4105
4106
4107
4108
4109
4110
4111
4112
4113
4114
4115
4116
4117
4118
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
4150
4151
4152
4153
4154
4155
4156
4157
4158
4159
4160
4161
4162
4163
4164
4165
4166
4167
4168
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
4200
4201
4202
4203
4204
4205
4206
4207
4208
4209
4210
4211
4212
4213
4214
4215
4216
4217
4218
4219
4220
4221
4222
4223
4224
4225
4226
4227
4228
4229
4230
4231
4232
4233
4234
4235
4236
4237
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
4250
4251
4252
4253
4254
4255
4256
4257
4258
4259
4260
4261
4262
4263
4264
4265
4266
4267
4268
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
4300
4301
4302
4303
4304
4305
4306
4307
4308
4309
4310
4311
4312
4313
4314
4315
4316
4317
4318
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
4350
4351
4352
4353
4354
4355
4356
4357
4358
4359
4360
4361
4362
4363
4364
4365
4366
4367
4368
4369
4370
4371
4372
4373
4374
4375
4376
4377
4378
4379
4380
4381
4382
4383
4384
4385
4386
4387
4388
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
4400
4401
4402
4403
4404
4405
4406
4407
4408
4409
4410
4411
4412
4413
4414
4415
4416
4417
4418
4419
4420
4421
4422
4423
4424
4425
4426
4427
4428
4429
4430
4431
4432
4433
4434
4435
4436
4437
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
4450
4451
4452
4453
4454
4455
4456
4457
4458
4459
4460
4461
4462
4463
4464
4465
4466
4467
4468
4469
4470
4471
4472
4473
4474
4475
4476
4477
4478
4479
4480
4481
4482
4483
4484
4485
4486
4487
4488
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499
4500
4501
4502
4503
4504
4505
4506
4507
4508
4509
4510
4511
4512
4513
4514
4515
4516
4517
4518
4519
4520
4521
4522
4523
4524
4525
4526
4527
4528
4529
4530
4531
4532
4533
4534
4535
4536
4537
4538
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549
4550
4551
4552
4553
4554
4555
4556
4557
4558
4559
4560
4561
4562
4563
4564
4565
4566
4567
4568
4569
4570
4571
4572
4573
4574
4575
4576
4577
4578
4579
4580
4581
4582
4583
4584
4585
4586
4587
4588
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4599
4600
4601
4602
4603
4604
4605
4606
4607
4608
4609
4610
4611
4612
4613
4614
4615
4616
4617
4618
4619
4620
4621
4622
4623
4624
4625
4626
4627
4628
4629
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
4650
4651
4652
4653
4654
4655
4656
4657
4658
4659
4660
4661
4662
4663
4664
4665
4666
4667
4668
4669
4670
4671
4672
4673
4674
4675
4676
4677
4678
4679
4680
4681
4682
4683
4684
4685
4686
4687
4688
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4699
4700
4701
4702
4703
4704
4705
4706
4707
4708
4709
4710
4711
4712
4713
4714
4715
4716
4717
4718
4719
4720
4721
4722
4723
4724
4725
4726
4727
4728
4729
4730
4731
4732
4733
4734
4735
4736
4737
4738
4739
4740
4741
4742
4743
4744
4745
4746
4747
4748
4749
4750
4751
4752
4753
4754
4755
4756
4757
4758
4759
4760
4761
4762
4763
4764
4765
4766
4767
4768
4769
4770
4771
4772
4773
4774
4775
4776
4777
4778
4779
4780
4781
4782
4783
4784
4785
4786
4787
4788
4789
4790
4791
4792
4793
4794
4795
4796
4797
4798
4799
4800
4801
4802
4803
4804
4805
4806
4807
4808
4809
4810
4811
4812
4813
4814
4815
4816
4817
4818
4819
4820
4821
4822
4823
4824
4825
4826
4827
4828
4829
4830
4831
4832
4833
4834
4835
4836
4837
4838
4839
4840
4841
4842
4843
4844
4845
4846
4847
4848
4849
4850
4851
4852
4853
4854
4855
4856
4857
4858
4859
4860
4861
4862
4863
4864
4865
4866
4867
4868
4869
4870
4871
4872
4873
4874
4875
4876
4877
4878
4879
4880
4881
4882
4883
4884
4885
4886
4887
4888
4889
4890
4891
4892
4893
4894
4895
4896
4897
4898
4899
4900
4901
4902
4903
4904
4905
4906
4907
4908
4909
4910
4911
4912
4913
4914
4915
4916
4917
4918
4919
4920
4921
4922
4923
4924
4925
4926
4927
4928
4929
4930
4931
4932
4933
4934
4935
4936
4937
4938
4939
4940
4941
4942
4943
4944
4945
4946
4947
4948
4949
4950
4951
4952
4953
4954
4955
4956
4957
4958
4959
4960
4961
4962
4963
4964
4965
4966
4967
4968
4969
4970
4971
4972
4973
4974
4975
4976
4977
4978
4979
4980
4981
4982
4983
4984
4985
4986
4987
4988
4989
4990
4991
4992
4993
4994
4995
4996
4997
4998
4999
5000

```

procedure calvariable(var alldone: boolean);

(\* The procedure performs the following operations:

- skips all the empty lines and the lines with only blanks.
- checks if the first token equals to 'ENTRY' or 'URG'.
- if the first token equals to 'ENTRY' means that the following lines have opcodes and so there are no more variables for analysis,so it sets the variable alldone to true.
- if the first token equals to 'URG' then it calculates the following address of the URG and it checks if the last declared URG had higher address and in this case it prints an errormessage.
- if the first token is not equal to ORG or ENTRY then
  - calls the procedure settype to find the type of the variable (BYTE,WORD,LONG,ARRAY,LABEL ) and checks if the variable has already been declared or the variable name is a reserved word or constant.
  - if the token is label it checks if there is room in the table for the new label and if it is it makes an entry.
  - if the token is an array then it calls the procedure re getdimension to find the type of the array and the dimension of the array.
  - in either case of variable or array it searches for the intial values .If it is array it checks also if the values are separated with ',' and if there are less or more than the size of the array dimension
  - if an error occurs it prints an errormessage and clears the input buffer.

\*)

```

3125 var
3126 times, checknum, sizevar, kind, start: integer;
3127 value: real;
3128 token: one;
3129 correct: boolean;
3130 unassigned: boolean;
3131 (* this variable permits the user to define
3132    only part of the initial values of an array*)
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3160
3161
3162
3163
3164
3165
3166
3167
begin (*1*)

sizevar := 1;
unassigned := false;
correct := true;
skipcomments;
gettoken(5, token, correct);
(* gets the first token *)
if (token <> 'EN[RY'] and correct and (token <> 'ORG') (*2*))
then begin
    gettype(kind, correct);
    (* find the type of the variable *)
    (* types 2= byte, 4 = word, 8 = long, 0 = array, 10 = label*)
    if correct then begin (* 2b*)
        check(correct, token, 2);
        (* checks if the name of the variable has been
           already declared or is a reserved word or constant*)
        if kind = 10 then
            checknum := 3
        else
            checknum := 2;
        if correct then begin (* 3a)
            if checknum = 3 then begin (*4a)
                (* means that it is label *)

```

```

3181
3182
3183
3184
3185
3186
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3220
3221
3222

if labelindex < maxlabel then begin (*5*)
    labelindex := labelindex + 1;
    variable[labelindex].name := token;
    (* makes an entry in the table *)
    if ch <> last then
        skipblanks(2)
    (* checks if they follow any invalid characters *)
end else
    errormessage(14)
end else begin (* 4*) (* 5*)
    if varindex < maxvar then begin (* 6*)
        (* no label so other type variable or array *)
        varindex := varindex + 1;
        variable[varindex].name := token;
        if kind = 0 then begin (* 7*)
            getdimension(sizevar, kind, correct);
            (* if variable then checks for her dimension
               and his type BYTE - WORD - LONG *)
            load;
            if not correct then
                errormessage(16)
            end; (*7*)
            if correct then begin (*8*)
                getassig(correct);
                (* finds the ':'= symbol *)
                if correct then begin (* 9*)
                    load;
                    with variable[varindex] do begin (* 10*)
                        size := kind;
                        initial := PC;
                        final := PC + kind * sizevar div 2 - 1;
                        if kind = 2 then (* byte *)
                            final := final + kind * sizevar div 2;
                        end; (* 10*)
                        (* makes the entries in the table as

```

```

3245 name , size , initial address and final
3246 address *)
3247 times := 0;
3248 start := PC;
3249 repeat
3250 if not unassigned then (* else repeats
3251 to use the last evaluated value *)
3252 evalconst(value, correct, kind);
3253 (* the loop is executed once if it is
3254 variable and so many times as the di-
3255 mension of the array *)
3256 (* if in the user in the array writes value
3257 and then . the program fills the rest
3258 of the array with the last value *)
3259 if correct then begin
3260 if kind <> 2 then
3261 setmem(start, value, kind)
3262 else
3263 setmem (start,value * 256 , 2 * kind );
3264 (* case of byte it sets only the high byte*)
3265 (* transfers the read value of the variable
3266 or each element of the array in the virtual
3267 memory *)
3268 end;
3269 start := start + kind div 2;
3270 if kind = 2 then
3271 start := start + kind div 2;
3272 PC := start;
3273 times := times + 1;
3274 if (times < sizevar) and not unassigned then begin
3275 skinblanks(0);
3276 (* checks for ', ' *)
3277 if (ch <> ',') and (ch <> '.') then
3278 errormessage (53);
3279 if ch = '.', then

```



```

3301
3302
3303
3304
3305
3306
3307
3308
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342

        unassigned := true;
        load;
        skipblanks(0)
        (* skips all the blanks *)

    end;
    if (ch = last) and (times < sizevar)
        and not unassigned then begin
        clear;
        (* if the array is big then the initial
           values may follow in the next line *)
        ch := ' ';
    end
    until (times >= sizevar) or not correct; (*9*)
    if (ch <> last) and correct then
        skipblanks(2);
    (* checks for invalid characters after the
       the values (s) *)
    if times < sizevar then
        errormessage(22)
    end
    end; (* 8*)
    if not correct then
        retreat(1);
    end else
        (*6*) (* if varindex < 39 *)
        errormessage(15)
    end (* 5*)
    end (* 3 *)
    end (* 2b *)
    else
        errormessage(40);
        (* no correct type declaration was encountered*)
        clear
    end else begin (* 2 *) (* 2a *)
        (* OKG statement *)
        if correct then begin (* 30 *)

```

```

3364 if token = 'ORG' then
3365   begin (*31*)
3366     evalconst(value,correct,4);
3367     if odd (trunc(value)) then
3368       value := value + 1;
3369     (* it compensates for user error with
3370       odd address *)
3371     (* it calculates the following address*)
3372     if correct then begin (*32*)
3373       if (LAST < value) and (PC <= value) then
3374         (* checks if the last declared ORG
3375           had higher address or also for the current PC *)
3376         begin (*33*)
3377           PC := trunc(value);
3378           LAST := value;
3379           (* update the address pointer of
3380             the ORG's *)
3381           end else (*33*)
3382             errormessage(36);
3383         end;(*32*)
3384       clear;
3385     end else (*31*)
3386       alldone := true
3387     end else (*30*)
3388       clear
3389     end (*28*) (* calvariable *)
3390     end; (* calvariable *)
3391
3392
3393
3394
3395
3396
3397
3398
3399
3400
3401
3402
3403
3404
3405
3406
3407

```

```

3421
3422
3423
3424
3425
3426
3427
3428
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
3450
3451
3452
3453
3454
3455
3456
3457
3458
3459
3460
3461
3462

procedure varanalyze;
(*This procedure is used to calculate the variables
calling repeatedly the procedure calvariable.
The procedure terminates if the procedure calvariable
returns the parameter alldone set to true.
In order the procedure calls for the first time the
procedure calvariable it must encountered the token
VARIA.
If the procedure calvariable returns alldone equal to
true then retreats the character pointer in the input
buffer in the first character in order to be analyzed
by the opcode analyzer *)

var
correct, alldone: boolean;

begin
alldone := false;
skipcomments;
(* skips blanks lines and lines with comments *)
gettoken(5, token, correct);
if token = 'VARIA' then begin
clear;
repeat
skipcomments;
calvariable(alldone)
until alldone
(* alldone is true if the reserved word 'FNTRY' is
encountered *)
end else
kit;
(* retreats the character pointer of the

```

```
3485      inout buffer *)
3486  if alldone then
3487      kit
3488  end; (* varanalyze *)
```

```
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
3500
3501
3502
3503
3504
3505
3506
3507
3508
3509
3510
3511
3512
3513
3514
3515
3516
3517
3518
3519
3520
3521
3522
3523
3524
3525
3526
3527
```

```

3541
3542
3543
3544
3545 procedure zerooperands(index: integer; var correct: boolean);
3546
3547 (* This procedure is used to calculate the opcodes
3548    with zero operands so it retrieves their value
3549    from the table of the opcodes and it sets the
3550    virtual memory *)
3551
3552 var
3553     value: real;
3554
3555 begin
3556     value := opcode[index].base;
3557     (* HALI, IREF, NOP *)
3558     setmem(PC, value, 4);
3559     correct := true;
3560     PC := PC + 2
3561 end; (* zerooperands *)
3562
3563
3564
3565
3566
3567
3568
3569
3570
3571
3572
3573
3574
3575
3576
3577
3578
3579
3580
3581
3582

```

```

3605 procedure setlistoflabels(index: integer);
3606
3607 (* This procedure is used in the case that a label
3608    is referred forward so it finds the position of
3609    of the label in the label table and inserts the
3610    address where the reference has been done .A
3611    label by this method can be forward referred 20
3612    times and this feature permits the assembler to
3613    make only one pass *)
3614
3615 var
3616   i: integer;
3617
3618 begin
3619   i := 0;
3620   while (labelist(index, i) <> -1) and (i <= 20) do
3621     i := i + 1;
3622   (* searches for an empty position in the table*)
3623   if i <= 20 then
3624     labelist(index, i) := pf + 2
3625   (* inserts the reference position *)
3626   else
3627     errormessage(27)
3628   end; { setlistoflabels }

```

```

3605
3606
3607
3608
3609
3610
3611
3612
3613
3614
3615
3616
3617
3618
3619
3620
3621
3622
3623
3624
3625
3626
3627
3628
3629
3630
3631
3632
3633
3634
3635
3636
3637
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647

```

```

3661 procedure findccflags(token: one ; var kind, index : real;
3662 var correct : boolean);
3663
3664 (* This procedure is used to find if an operand token is
3665 either cc (condition code) or control register.
3666 In either case it sets the parameter kind to the value
3667 5 if the token is cc or 4 if the token is control regis-
3668 ter.
3669 Also the parameter index is set with the index of the
3670 array where the token was found
3671 If the token is not found then it returns false with
3672 the parameter correct.
3673 The variable alldone is used to terminate the search
3674 if the token is found in one of the two tables *)
3675
3676 var
3677 k: integer;
3678 alldone: boolean;
3679
3680 begin
3681 alldone := false;
3682 k := -1;
3683 repeat
3684 k := k + 1
3685 until (controlreg[k] = token) or (k = 6);
3686 (* checks for control register *)
3687 if controlreg[k] = token then begin
3688 kind := 4;
3689 index := k;
3690 alldone := true
3691 end;
3692 if not alldone then begin
3693 k := 0;
3694 repeat
3695 k := k + 1

```



```

3725 until (condition[k] = token) or (k = 15);
3726 (* checks for condition code *)
3727 if condition[k] = token then begin
3728   kind := 5;
3729   index := k;
3730   alldone := true
3731   end
3732 end;
3733 correct := alldone
3734 end; { findccflags }
3735
3736
3737
3738
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749
3750
3751
3752
3753
3754
3755
3756
3757
3758
3759
3760
3761
3762
3763
3764
3765
3766
3767

```

```

3781
3782
3783
3784
3785 procedure findkindoftoken(token: one; var kind, index,
3786 reftype: real; var correct: boolean);
3787 (* This procedure is similar to the procedure findccflans
3788 but because the occurrence of the variables, registers, labels
3789 and the constants is more frequently it is called first to
3790 find if the token is one of the above mentioned types.
3791 The sequence of search is the following :
3792 - registers
3793 - variables - labels
3794 - constants
3795 If the token is found in one of the tables then the procedure
3796 terminates the search setting the variable alldone to true.
3797 If the token is found then it returns :
3798 parameter correct equal to true
3799 parameter kind :
3800 0 if it is register
3801 1 if it is variable
3802 2 if it is label
3803 3 if it is constant
3804 parameter index equal to the index of the table where
3805 the token was found
3806 parameter reftype equal to the length of the register,
3807 or the address of the variable or the label or the value
3808 of the constant.
3809 If the token does not have as first character R the pro
3810 cedure does not search the registers table.
3811 *)
3812
3813 var
3814 alldone: boolean;
3815 k: integer;
3816
3817 begin
3818 k := 0;
3819
3820
3821

```

```

3845 alldone := false;
3846 if token[0] = 'R' then begin (* check for register*)
3847   k := -1;
3848   repeat
3849     k := k + 1
3850   until (register[k].name = token) or (k = maxreg);
3851   end;
3852   (* checks for register *)
3853   if register[k].name = token then begin
3854     kind := 0;
3855     index := register[k].value;
3856     regtype := register[k].length;
3857     alldone := true
3858   end else begin
3859     (* no register so it continues the search *)
3860     k := -1;
3861     repeat
3862       k := k + 1
3863     until (variable[k].name = token) or (k >= varindex);
3864     (* search for variable *)
3865     if variable[k].name = token then begin
3866       kind := 1;
3867       index := k;
3868       regtype := variable[k].initial;
3869       alldone := true
3870     end;
3871     if not alldone then begin
3872       (* not found yet so it continues the search *)
3873       k := maxvar;
3874       repeat
3875         (* searches for label *)
3876           k := k + 1
3877         until (variable[k].name = token) or (k >= labelindex);
3878         if variable[k].name = token then begin
3879           kind := 2;

```

```

3901   index := k;
3902   regtype := variable[k].initial;
3903   alldone := true
3904 end
3905 end;
3906 if not alldone then begin
3907   (* not found yet so it continues for constant *)
3908   k := -1;
3909   repeat
3910     k := k + 1
3911   until (constlist[k].name = token) or (k = constindex);
3912   if constlist[k].name = token then begin
3913     kind := 3;
3914     index := k;
3915     regtype := constlist[k].value;
3916     alldone := true
3917   end
3918 end
3919 end;
3920 correct := alldone
3921 (* correct is true only if it is found in one of the tables*)
3922 end; { findkindoftoken }
3923
3924
3925
3926

```

```

procedure getoperand(typekind: current; var value, value1: real;
var which : permit; var correct: boolean);

```

```

(* This procedure is the main searching mechanism for the
token .The caller with the parameter typekind it pass
the subset of the permitted types of address for the
searched token and the procedure it returns :
- correct equal to true if the token was in the subset
and the evaluation of his value was correct.
- which equal to the type of the found token
- value equal to the number of the register in the
case that the token is register type or the address
of the token if it is label or variable or the value
of the token if it is constant or the index of the
table where was found if it is cc or control register.
- value1 equal to the length of the register or the number
of the register in the case of index addressing.

```

```

The procedure at first it checks if before the token is any
special character as %, #, $ @ or the token is decimal number
and then it checks what of token it is with the help of the
first special if it is existed.

```

```

For the cases of direct address it checks if it follows
indexed address and in the case of variables it checks if
follows subscript of an array.
The details of the procedure are described with comments
into the procedure *)

```

```

var
first: boolean;
index1, index2: real;
reftype, index, kind: real;
token: one;

```

```

3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
4000
4001
4002
4003
4004
4005
4006
4007

```

```

4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
4050
4051
4052
4053
4054
4055
4056
4057
4058
4059
4060
4061
4062

begin (*1*)
correct := true;
if ch = 'a' then begin (*2*)
(* checks if the token is indirect register *)
  which := ir;
  load
end else if ch = '#' then begin (*2*) (*3*)
(* checks if the token is immediate data *)
  which := im;
  load
end else if ch = 'b' then begin (* so decimal,hex,' string,*) (*2*)
(* checks if the token is relative address *)
  which := ra;
  first := true;
  load;
  skipblanks(0);
  if ch = '+' then
    load
  else
    begin
      errormessage (38);
      correct := false;
    end;
end else begin (*4*) (*5*)
  if ch = '%' then begin (*6*)
    (* so the token may be direct address with hex characters *)
    which := da;
    hexvalue(value, correct, R)
    (* if the token is hex value then it calls the procedure
       hexvalue to calculate the address *)
  end else if ch in ['0'..'9'] then begin (*6*) (*7*)
    (* checks if the token is decimal number so direct address*)

```



```

4084 which := da;
4085 decimal(value, correct, 8)
4086 (* if the token is decimal value then it calls the
4087 procedure decimal to calculates the decimal value *)
4088 end else begin (*7*) (*8*)
4089 (* variable, label, register*)
4090 (* or CC, control register*)
4091 gettoken(5, token, correct);
4092 (* calls the procedure gettoken to return the token *)
4093 findkindoftoken(token, kind, index, regtype, correct);
4094 (* checks if the token is variable, register, constant
4095 or label *)
4096 if not correct and ((control in typekind) or (cc in typekind))
4097 then
4098 (* it restricts the search if the kind that is expected
4099 is not control register or cc *)
4100 findccflags(token, kind, index, correct);
4101 (* checks if the token is control register or cc code *)
4102 if correct then
4103 (* if the token was found then it assigns to the variable
4104 which what was the token *)
4105 case trunc(kind) of
4106 0:
4107 1: which := reg;
4108 begin
4109 value := regtype;
4110 which := da
4111 end; (* variable*)
4112 2:
4113 begin
4114 value := regtype;
4115 which := da
4116 end; (*label *)
4117 3:

```



```

4141
4142
4143
4144
4145
4146
4147
4148
4149
4150
4151
4152
4153
4154
4155
4156
4157
4158
4159
4160
4161
4162
4163
4164
4165
4166
4167
4168
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4180
4181
4182

begin := reatype;
value := da;
which := const *
end;

4: which := control; (*control register*)
5: which := cc
end (* cc condition register *);
if (kind = 2) and (ra in typekind) then
    which := ra
end
(* the opcode makes the relative addressing and not the
   syntax of the operand *)
end (*8*); (*5*)
if correct then begin (*10*)
    if not (which in typekind) and (which <> xi) then
        (* it checks if the token is in the passed subset of
           the caller *)
        correct := false;
    if correct then begin (*11*)
        (* starts the calculations depending on the type of the
           token *)
        case which of
            control :
                begin
                    value := index;
                end;
            reg:
                begin
                    value := index;
                    value1 := reatype
                end;
            ir:
                begin (*12*)

```

```

4204 gettoken(5, token, correct);
4205 (* after the the if it follows an register *)
4206 if correct then begin (*13*)
4207   findkindoftoken(token, kind, index, reatype, correct);
4208   if (kind = 0) and correct then begin (*14*)
4209     value := index;
4210     (* is correct only if the token is equal to register*)
4211     value1 := reatype
4212   end else (*14*)
4213     correct := false
4214   end
4215 end; (*13*) (*12*)
4216 cc:
4217 begin (*15*)
4218   value := index
4219 end; (*15*)
4220 im:
4221 begin (*16*)
4222   (* after the # sign it may follow address or
4223     string or hex characters *)
4224     (* decimal, hex, string', constant, variable*)
4225     if ch = '%', then
4226       hexvalue(value, correct, 8)
4227     else if ch = chr(39) then
4228       (* it follows string *)
4229       string(value, correct, 8)
4230     else if ch in ['0'..'9'] then
4231       decimal(value, correct, 8)
4232       (* it follows decimal value *)
4233     else begin (*17*)
4234       gettoken(5, token, correct);
4235       (* else it is variable or constant *)
4236       if correct then
4237         findkindoftoken(token, kind, index, reatype, correct);
4238         if correct and (kind <> 2) and (kind <> 0) then

```

```

4261
4262
4263
4264
4265
4266
4267
4268
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
4300

(*kind = 0 is reg, kind = 2 is label *)
(* it is not possible to be register or label *)
    value := reatype
else
    correct := false
end
end; (*17*) (*16*)
da:
begin (*20*)
    if kind = 2 then begin
        if value < 0 then begin
            value := 0;
            setlistoflabels(trunc(index))
            (* the token is label not found yet so
               it saves the address where was encountered *)
        end
    end
end;
ra:
begin (*21*)
    if not first then begin (* 22 *)
        if reatype < 0 then begin (* 23 *)
            (* means no encountered yet label *)
                value := 0;
                setlistoflabels(trunc(index))
            (* it is label that is not encountered yet *)
        end else (* 23 *)
            value := reatype
        end else if correct then begin (*28*)
            if ch = '%' then
                (* it is hex value *)
                    hexvalue(value, correct, 8)
            else if ch in ['0'..'9'] then
                (* it is decimal value *)
                    decimal(value, correct, 8)
            end
        end
    end
end

```

```

4325 else begin (* 29 *)
4326   gettoken(5, token, correct);
4327   findkindoftoken(token, kind, index, reatype, correct);
4328   if correct and ((kind = 1) or (kind = 3)) then
4329     (* constant or variable *)
4330     value := reatype
4331   else
4332     correct := false
4333   end; (* 29 *)
4334   value := value - 2
4335   end (* 28 *)
4336   end (* 21 *)
4337   end (* case *)
4338   end (* 11 *)
4339   end; (* 10 *)
4340   if (which = da) and correct then begin (* 60 *)
4341     (* the direct address may be followed by indirect one *)
4342     skipblanks(0);
4343     if (ch = '|') and (kind = 1) then begin
4344       (* it is variable and then it follows the subscript *)
4345       (* case of variable , it looks for array index *)
4346       (* 61 *)
4347       load;
4348       skipblanks(0);
4349       decimal(index1, correct, 2);
4350       (* calculates the index of the array *)
4351       if correct then begin (* 62 *)
4352         index2 := variable[trunc(index)].initial
4353           + trunc(index1) * variable[trunc(index)].size / 2;
4354         (* checks if the subscript is in the bounds of the a
4355            array dimension *)
4356         if (index2 <= variable[trunc(index)].final
4357            - variable[trunc(index)].size ) then
4358           value := index2
4359         else begin (* 63 *)
4360
4361
4362
4363
4364
4365
4366
4367

```

```

4381      errormessage(29);
4382      correct := false
4383    end
4384  end (*b3*); (*b2*)
4385  skipblanks(0);
4386  if correct then begin (*b4*)
4387    if ch = ']' then
4388      (* the subscript of the array must be enclosed in []*)
4389      load
4390    else begin (*b5*)
4391      errormessage(30);
4392      correct := false
4393    end (*b5*)
4394  end (*b4*)
4395  end else if ch = '(' then begin (*b1*) (*b6*)
4396    (* so may be index register *)
4397    load;
4398  skipblanks(0);
4399  gettoken(5, token, correct);
4400  (* get the next token *)
4401  if correct then
4402    findkindoftoken(token, kind, index, regtype, correct);
4403    (* it checks the type of the token *)
4404    if correct and (kind = 0) then begin (*b7*)
4405      which := xi;
4406      (* only if it is register type is permitted *)
4407      value1 := index;
4408      if regtype <> 1 then begin
4409        errormessage(26);
4410        (* the index registers must always be word registers*)
4411        correct := false
4412      end;
4413      skipblanks(0);
4414      if ch = ')' then
4415        load

```

```
4444 else begin (*68*)
4445 correct := false;
4446 errormessage(31)
4447 end
4448
4449 end else (*68*)
4450 errormessage(32)
4451 end (*67*)
4452 end; (*60*) (*61*)
4453 end; ( notonerand )
4454
4455
4456
4457
4458
4459
4460
4461
4462
4463
4464
4465
4466
4467
4468
4469
4470
4471
4472
4473
4474
4475
4476
```

```

program assem (input,output);
#include 'assembler.i'

```

(\*\*\*\*\*)

The procedures that follow are the procedures that encode the different types of opcodes and they are divided in categories depending on the number of the operands of the opcodes. So there are the following procedures :

- oneoperands
- twooperands (this category is divided in twoop1, twoop2,twoop3,twoop4 due to the big number of opcodes with two operands.
- threeoperands
- fouroperands

All the procedures use the same names of variables that are local to each procedure and this facilitates the reading of the code. The variables that are used are the following:

- which2,which3,which4 : this type of the variable is used in the calling of the procedure getoperand as a parameter and the the procedure getoperand it returns the type of the token that was found.
- value, value2,value4 : this type of variable is used as parameter in the calling of the procedure getoperand and the procedure getoperand returns the address of the variable or label or the value of a constant or the num



- ber of a register or the index of the table where was found a cc code or a control register.
- value1,value3,value5 : this type of parameter is also used as a parameter and the procedure getoperand returns the length of the register or the register in the case of index addressing.
- type..type20 : this type of variable is used again as parameter and the calling procedure it passes the permitted types of operands as a subset
- correct : is also used as a parameter and is used by the procedure getoperand to define if the token was correct and it was in the subset that was passing by the caller with the parameter type..type20
- final ,second , third : this types of the variables are used to calculate the value with that the memory will be set after the completion of the encoding .(final if the memory will be set with one word , second if it will be set also with a second word and third if it will be set with third word as in the case of LDM or STORE ( Note the number of the variables that are used by each procedure depends on the number of the operands)

The base algorithm that is used from the procedures is the same as follows :

- the opcodes are divided in subcategories depending on the permitted type of operands for the first and the second and so operand .
- the caller pass to the procedure getoperand the subset of the permitted types of operands
- the procedure getoperand returns the values of the operand as was previously discussed and also if it is correct or not.
- the caller continues to call the procedure getoperand till to find all the operands or an incorrect operand and then it terminates the search.
- the caller if it gets all the operands it checks for

validity for example word register with word opcode  
 - it also calculates the encoding depending on the base  
 value of the opcode that exists in a table and on  
 the other parameters that affect the value as the  
 addressing mode and the values of the destination  
 and the source operand.

The above algorithm is the main one with exceptions so in  
 each procedure that follows there are provided comments in  
 the code that describe all the differents  
 Because all the opcodes are refered depending on their  
 value in the opcode table that is loaded in the booting  
 of the program the two major tables opcode table and  
 register table are listed :

# OPCODE TABLE

| opcode | No of<br>operands | base value |
|--------|-------------------|------------|
| HALT   | 0                 | 31232      |
| TRFT   | 0                 | 31488      |
| NOP    | 0                 | 36103      |
| DAR    | 1                 | 45056      |
| EXTS   | 1                 | 45322      |
| EXTSR  | 1                 | 45312      |
| EXTSL  | 1                 | 45319      |
| RET    | 1                 | 40448      |
| SC     | 1                 | 32512      |
| NEG    | 1                 | 3330       |
| NEGB   | 1                 | 3074       |
| CUM    | 1                 | 3328       |
| CUMB   | 1                 | 3072       |



|     |        |   |       |
|-----|--------|---|-------|
| 241 | LDK    | 2 | 48384 |
| 242 | LDAR   | 2 | 13312 |
| 243 | LDA    | 2 | 30208 |
| 244 | RLDB   | 2 | 48640 |
| 245 | RKDB   | 2 | 48128 |
| 246 | RLR    | 2 | 45576 |
| 247 | RL     | 2 | 45832 |
| 248 | RLCB   | 2 | 45568 |
| 249 | RLC    | 2 | 45824 |
| 250 | RRR    | 2 | 45580 |
| 251 | RR     | 2 | 45836 |
| 252 | RRCH   | 2 | 45572 |
| 253 | RRC    | 2 | 45828 |
| 254 | SDAB   | 2 | 45579 |
| 255 | SDA    | 2 | 45835 |
| 256 | SDAL   | 2 | 45839 |
| 257 | SDIB   | 2 | 45571 |
| 258 | SDI    | 2 | 45827 |
| 259 | SDL    | 2 | 45831 |
| 260 | DECB   | 2 | 10752 |
| 261 | DEC    | 2 | 11008 |
| 262 | TNCB   | 2 | 10240 |
| 263 | TNC    | 2 | 10496 |
| 264 | RITB   | 2 | 9728  |
| 265 | RIT    | 2 | 9984  |
| 266 | RESB   | 2 | 8704  |
| 267 | RES    | 2 | 8960  |
| 268 | SETB   | 2 | 9216  |
| 269 | SET    | 2 | 9472  |
| 270 | STORF  | 2 | 12032 |
| 271 | STORFB | 2 | 11776 |
| 272 | STORFL | 2 | 7424  |
| 273 | PUP    | 2 | 5888  |
| 274 | POPL   | 2 | 5376  |
| 275 | PUSH   | 2 | 4864  |

|     |        |   |       |
|-----|--------|---|-------|
| 304 |        |   |       |
| 305 | PUSHI  | 2 | 4352  |
| 306 | JP     | 2 | 7680  |
| 307 | JK     | 2 | 57344 |
| 308 | TCCB   | 2 | 44544 |
| 309 | TCC    | 2 | 44800 |
| 310 | LDRB   | 2 | 12288 |
| 311 | LDR    | 2 | 12544 |
| 312 | LDRL   | 2 | 13568 |
| 313 | CLR    | 1 | 3336  |
| 314 | CLRB   | 1 | 3080  |
| 315 | LDM    | 3 | 7168  |
| 316 | DJNZ   | 2 | 61568 |
| 317 | DBJN7  | 2 | 61440 |
| 318 | SLAB   | 2 | 45577 |
| 319 | SLA    | 2 | 45833 |
| 320 | SLAL   | 2 | 45837 |
| 321 | SLLB   | 2 | 45569 |
| 322 | SLL    | 2 | 45825 |
| 323 | SLLL   | 2 | 45829 |
| 324 | SRAB   | 2 | 45577 |
| 325 | SRA    | 2 | 45833 |
| 326 | SRAL   | 2 | 45837 |
| 327 | SRIB   | 2 | 45569 |
| 328 | SRL    | 2 | 45825 |
| 329 | SRLI   | 2 | 45829 |
| 330 | INB    | 2 | 15360 |
| 331 | IN     | 2 | 15616 |
| 332 | OUTB   | 2 | 15872 |
| 333 | OUT    | 2 | 16128 |
| 334 | LDCLB  | 2 | 35840 |
| 335 | LDCHI  | 2 | 32000 |
| 336 | COMFLG | 2 | 36101 |
| 337 | RESFLG | 2 | 36097 |
| 338 | SETFLG | 2 | 36099 |
| 339 | LPDS   | 1 | 14592 |
| 340 |        |   |       |
| 341 |        |   |       |
| 342 |        |   |       |
| 343 |        |   |       |
| 344 |        |   |       |
| 345 |        |   |       |
| 346 |        |   |       |
| 347 |        |   |       |

|     |    |  |  |  |  |
|-----|----|--|--|--|--|
| 361 |    |  |  |  |  |
| 362 |    |  |  |  |  |
| 363 |    |  |  |  |  |
| 364 |    |  |  |  |  |
| 365 |    |  |  |  |  |
| 366 |    |  |  |  |  |
| 367 |    |  |  |  |  |
| 368 |    |  |  |  |  |
| 369 |    |  |  |  |  |
| 370 |    |  |  |  |  |
| 371 |    |  |  |  |  |
| 372 |    |  |  |  |  |
| 373 |    |  |  |  |  |
| 374 |    |  |  |  |  |
| 375 |    |  |  |  |  |
| 376 |    |  |  |  |  |
| 377 |    |  |  |  |  |
| 378 |    |  |  |  |  |
| 379 |    |  |  |  |  |
| 380 |    |  |  |  |  |
| 381 |    |  |  |  |  |
| 382 |    |  |  |  |  |
| 383 |    |  |  |  |  |
| 384 |    |  |  |  |  |
| 385 |    |  |  |  |  |
| 386 |    |  |  |  |  |
| 387 |    |  |  |  |  |
| 388 |    |  |  |  |  |
| 389 |    |  |  |  |  |
| 390 |    |  |  |  |  |
| 391 |    |  |  |  |  |
| 392 |    |  |  |  |  |
| 393 |    |  |  |  |  |
| 394 |    |  |  |  |  |
| 395 |    |  |  |  |  |
| 396 |    |  |  |  |  |
| 397 |    |  |  |  |  |
| 398 |    |  |  |  |  |
| 399 |    |  |  |  |  |
| 400 |    |  |  |  |  |
| 401 |    |  |  |  |  |
| 402 |    |  |  |  |  |
| 403 |    |  |  |  |  |
| 404 |    |  |  |  |  |
| 405 |    |  |  |  |  |
| 406 |    |  |  |  |  |
| 407 |    |  |  |  |  |
| 408 |    |  |  |  |  |
| 409 |    |  |  |  |  |
| 410 |    |  |  |  |  |
| 411 |    |  |  |  |  |
| 412 |    |  |  |  |  |
| 413 |    |  |  |  |  |
| 414 |    |  |  |  |  |
| 415 |    |  |  |  |  |
| 416 |    |  |  |  |  |
| 417 |    |  |  |  |  |
| 418 |    |  |  |  |  |
| 419 |    |  |  |  |  |
| 420 |    |  |  |  |  |
| 421 |    |  |  |  |  |
| 422 |    |  |  |  |  |
| 423 |    |  |  |  |  |
| 424 |    |  |  |  |  |
| 425 |    |  |  |  |  |
| 426 |    |  |  |  |  |
| 427 |    |  |  |  |  |
| 428 |    |  |  |  |  |
| 429 |    |  |  |  |  |
| 430 |    |  |  |  |  |
| 431 |    |  |  |  |  |
| 432 |    |  |  |  |  |
| 433 |    |  |  |  |  |
| 434 |    |  |  |  |  |
| 435 |    |  |  |  |  |
| 436 |    |  |  |  |  |
| 437 |    |  |  |  |  |
| 438 |    |  |  |  |  |
| 439 |    |  |  |  |  |
| 440 |    |  |  |  |  |
| 441 |    |  |  |  |  |
| 442 |    |  |  |  |  |
| 443 |    |  |  |  |  |
| 444 |    |  |  |  |  |
| 445 |    |  |  |  |  |
| 446 |    |  |  |  |  |
| 447 |    |  |  |  |  |
| 448 |    |  |  |  |  |
| 449 |    |  |  |  |  |
| 450 |    |  |  |  |  |
| 451 |    |  |  |  |  |
| 452 |    |  |  |  |  |
| 453 |    |  |  |  |  |
| 454 |    |  |  |  |  |
| 455 |    |  |  |  |  |
| 456 |    |  |  |  |  |
| 457 |    |  |  |  |  |
| 458 |    |  |  |  |  |
| 459 |    |  |  |  |  |
| 460 |    |  |  |  |  |
| 461 |    |  |  |  |  |
| 462 |    |  |  |  |  |
| 463 |    |  |  |  |  |
| 464 |    |  |  |  |  |
| 465 |    |  |  |  |  |
| 466 |    |  |  |  |  |
| 467 |    |  |  |  |  |
| 468 |    |  |  |  |  |
| 469 |    |  |  |  |  |
| 470 |    |  |  |  |  |
| 471 |    |  |  |  |  |
| 472 |    |  |  |  |  |
| 473 |    |  |  |  |  |
| 474 |    |  |  |  |  |
| 475 |    |  |  |  |  |
| 476 |    |  |  |  |  |
| 477 |    |  |  |  |  |
| 478 |    |  |  |  |  |
| 479 |    |  |  |  |  |
| 480 |    |  |  |  |  |
| 481 |    |  |  |  |  |
| 482 |    |  |  |  |  |
| 483 |    |  |  |  |  |
| 484 |    |  |  |  |  |
| 485 |    |  |  |  |  |
| 486 |    |  |  |  |  |
| 487 |    |  |  |  |  |
| 488 |    |  |  |  |  |
| 489 |    |  |  |  |  |
| 490 |    |  |  |  |  |
| 491 |    |  |  |  |  |
| 492 |    |  |  |  |  |
| 493 |    |  |  |  |  |
| 494 |    |  |  |  |  |
| 495 |    |  |  |  |  |
| 496 |    |  |  |  |  |
| 497 |    |  |  |  |  |
| 498 |    |  |  |  |  |
| 499 |    |  |  |  |  |
| 500 |    |  |  |  |  |
| 501 |    |  |  |  |  |
| 502 |    |  |  |  |  |
| 503 |    |  |  |  |  |
| 504 |    |  |  |  |  |
| 505 |    |  |  |  |  |
| 506 |    |  |  |  |  |
| 507 |    |  |  |  |  |
| 508 |    |  |  |  |  |
| 509 |    |  |  |  |  |
| 510 |    |  |  |  |  |
| 511 |    |  |  |  |  |
| 512 |    |  |  |  |  |
| 513 |    |  |  |  |  |
| 514 |    |  |  |  |  |
| 515 |    |  |  |  |  |
| 516 |    |  |  |  |  |
| 517 |    |  |  |  |  |
| 518 |    |  |  |  |  |
| 519 |    |  |  |  |  |
| 520 |    |  |  |  |  |
| 521 |    |  |  |  |  |
| 522 |    |  |  |  |  |
| 523 |    |  |  |  |  |
| 524 |    |  |  |  |  |
| 525 |    |  |  |  |  |
| 526 |    |  |  |  |  |
| 527 |    |  |  |  |  |
| 528 |    |  |  |  |  |
| 529 |    |  |  |  |  |
| 530 |    |  |  |  |  |
| 531 |    |  |  |  |  |
| 532 |    |  |  |  |  |
| 533 |    |  |  |  |  |
| 534 |    |  |  |  |  |
| 535 |    |  |  |  |  |
| 536 |    |  |  |  |  |
| 537 |    |  |  |  |  |
| 538 |    |  |  |  |  |
| 539 |    |  |  |  |  |
| 540 |    |  |  |  |  |
| 541 |    |  |  |  |  |
| 542 |    |  |  |  |  |
| 543 |    |  |  |  |  |
| 544 |    |  |  |  |  |
| 545 |    |  |  |  |  |
| 546 |    |  |  |  |  |
| 547 |    |  |  |  |  |
| 548 |    |  |  |  |  |
| 549 |    |  |  |  |  |
| 550 |    |  |  |  |  |
| 551 |    |  |  |  |  |
| 552 |    |  |  |  |  |
| 553 |    |  |  |  |  |
| 554 |    |  |  |  |  |
| 555 |    |  |  |  |  |
| 556 |    |  |  |  |  |
| 557 |    |  |  |  |  |
| 558 |    |  |  |  |  |
| 559 |    |  |  |  |  |
| 560 |    |  |  |  |  |
| 561 |    |  |  |  |  |
| 562 |    |  |  |  |  |
| 563 |    |  |  |  |  |
| 564 |    |  |  |  |  |
| 565 |    |  |  |  |  |
| 566 |    |  |  |  |  |
| 567 |    |  |  |  |  |
| 568 |    |  |  |  |  |
| 569 |    |  |  |  |  |
| 570 |    |  |  |  |  |
| 571 |    |  |  |  |  |
| 572 |    |  |  |  |  |
| 573 |    |  |  |  |  |
| 574 |    |  |  |  |  |
| 575 |    |  |  |  |  |
| 576 |    |  |  |  |  |
| 577 |    |  |  |  |  |
| 578 |    |  |  |  |  |
| 579 |    |  |  |  |  |
| 580 |    |  |  |  |  |
| 581 |    |  |  |  |  |
| 582 |    |  |  |  |  |
| 583 |    |  |  |  |  |
| 584 |    |  |  |  |  |
| 585 |    |  |  |  |  |
| 586 |    |  |  |  |  |
| 587 |    |  |  |  |  |
| 588 |    |  |  |  |  |
| 589 |    |  |  |  |  |
| 590 |    |  |  |  |  |
| 591 |    |  |  |  |  |
| 592 |    |  |  |  |  |
| 593 |    |  |  |  |  |
| 594 |    |  |  |  |  |
| 595 |    |  |  |  |  |
| 596 |    |  |  |  |  |
| 597 |    |  |  |  |  |
| 598 |    |  |  |  |  |
| 599 |    |  |  |  |  |
| 600 |    |  |  |  |  |
| 601 |    |  |  |  |  |
| 602 |    |  |  |  |  |
| 603 |    |  |  |  |  |
| 604 |    |  |  |  |  |
| 605 |    |  |  |  |  |
| 606 |    |  |  |  |  |
| 607 |    |  |  |  |  |
| 608 |    |  |  |  |  |
| 609 |    |  |  |  |  |
| 610 |    |  |  |  |  |
| 611 |    |  |  |  |  |
| 612 |    |  |  |  |  |
| 613 |    |  |  |  |  |
| 614 |    |  |  |  |  |
| 615 |    |  |  |  |  |
| 616 |    |  |  |  |  |
| 617 |    |  |  |  |  |
| 618 |    |  |  |  |  |
| 619 |    |  |  |  |  |
| 620 |    |  |  |  |  |
| 621 |    |  |  |  |  |
| 622 |    |  |  |  |  |
| 623 |    |  |  |  |  |
| 624 |    |  |  |  |  |
| 625 |    |  |  |  |  |
| 626 |    |  |  |  |  |
| 627 |    |  |  |  |  |
| 628 |    |  |  |  |  |
| 629 |    |  |  |  |  |
| 630 |    |  |  |  |  |
| 631 |    |  |  |  |  |
| 632 |    |  |  |  |  |
| 633 |    |  |  |  |  |
| 634 |    |  |  |  |  |
| 635 |    |  |  |  |  |
| 636 |    |  |  |  |  |
| 637 |    |  |  |  |  |
| 638 |    |  |  |  |  |
| 639 |    |  |  |  |  |
| 640 |    |  |  |  |  |
| 641 |    |  |  |  |  |
| 642 |    |  |  |  |  |
| 643 |    |  |  |  |  |
| 644 |    |  |  |  |  |
| 645 |    |  |  |  |  |
| 646 |    |  |  |  |  |
| 647 |    |  |  |  |  |
| 648 |    |  |  |  |  |
| 649 |    |  |  |  |  |
| 650 |    |  |  |  |  |
| 651 |    |  |  |  |  |
| 652 |    |  |  |  |  |
| 653 |    |  |  |  |  |
| 654 |    |  |  |  |  |
| 655 |    |  |  |  |  |
| 656 |    |  |  |  |  |
| 657 |    |  |  |  |  |
| 658 |    |  |  |  |  |
| 659 |    |  |  |  |  |
| 660 |    |  |  |  |  |
| 661 |    |  |  |  |  |
| 662 |    |  |  |  |  |
| 663 |    |  |  |  |  |
| 664 |    |  |  |  |  |
| 665 |    |  |  |  |  |
| 666 |    |  |  |  |  |
| 667 |    |  |  |  |  |
| 668 |    |  |  |  |  |
| 669 |    |  |  |  |  |
| 670 |    |  |  |  |  |
| 671 |    |  |  |  |  |
| 672 |    |  |  |  |  |
| 673 |    |  |  |  |  |
| 674 |    |  |  |  |  |
| 675 |    |  |  |  |  |
| 676 |    |  |  |  |  |
| 677 |    |  |  |  |  |
| 678 |    |  |  |  |  |
| 679 |    |  |  |  |  |
| 680 |    |  |  |  |  |
| 681 |    |  |  |  |  |
| 682 |    |  |  |  |  |
| 683 |    |  |  |  |  |
| 684 |    |  |  |  |  |
| 685 |    |  |  |  |  |
| 686 |    |  |  |  |  |
| 687 |    |  |  |  |  |
| 688 |    |  |  |  |  |
| 689 |    |  |  |  |  |
| 690 |    |  |  |  |  |
| 691 |    |  |  |  |  |
| 692 |    |  |  |  |  |
| 693 |    |  |  |  |  |
| 694 |    |  |  |  |  |
| 695 |    |  |  |  |  |
| 696 |    |  |  |  |  |
| 697 |    |  |  |  |  |
| 698 |    |  |  |  |  |
| 699 |    |  |  |  |  |
| 700 |    |  |  |  |  |
| 701 |    |  |  |  |  |
| 702 |    |  |  |  |  |
| 703 |    |  |  |  |  |
| 704 |    |  |  |  |  |
| 705 |    |  |  |  |  |
| 706 |    |  |  |  |  |
| 707 |    |  |  |  |  |
| 708 |    |  |  |  |  |
| 709 |    |  |  |  |  |
| 710 |    |  |  |  |  |
| 711 |    |  |  |  |  |
| 712 |    |  |  |  |  |
| 713 |    |  |  |  |  |
| 714 |    |  |  |  |  |
| 715 |    |  |  |  |  |
| 716 |    |  |  |  |  |
| 717 |    |  |  |  |  |
| 718 |    |  |  |  |  |
| 719 |    |  |  |  |  |
| 720 |    |  |  |  |  |
| 721 |    |  |  |  |  |
| 722 |    |  |  |  |  |
| 723 |    |  |  |  |  |
| 724 |    |  |  |  |  |
| 725 |    |  |  |  |  |
| 726 |    |  |  |  |  |
| 727 |    |  |  |  |  |
| 728 |    |  |  |  |  |
| 729 |    |  |  |  |  |
| 730 |    |  |  |  |  |
| 731 |    |  |  |  |  |
| 732 |    |  |  |  |  |
| 733 | </ |  |  |  |  |

|        |   |       |
|--------|---|-------|
| CPDR   | 4 | 47884 |
| CPTB   | 4 | 47616 |
| CPI    | 4 | 47872 |
| CPIRB  | 4 | 47620 |
| CPIR   | 4 | 47876 |
| CPSDR  | 4 | 47626 |
| CPSD   | 4 | 47882 |
| CPSURB | 4 | 47630 |
| CPSDR  | 4 | 47886 |
| CPSIR  | 4 | 47618 |
| CPSI   | 4 | 47874 |
| CPSIRB | 4 | 47622 |
| CPSIR  | 4 | 47878 |
| DI     | 1 | 31744 |
| EI     | 1 | 31748 |

# REGISTER TABLE

| register<br>name | value | length |
|------------------|-------|--------|
| P0               | 0     | 1      |
| R1               | 1     | 1      |
| R2               | 2     | 1      |
| R3               | 3     | 1      |
| R4               | 4     | 1      |
| R5               | 5     | 1      |
| R6               | 6     | 1      |
| P7               | 7     | 1      |
| R8               | 8     | 1      |
| R9               | 9     | 1      |
| R10              | 10    | 1      |
| P11              | 11    | 1      |
| R12              | 12    | 1      |

425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467



481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522

|      |    |   |
|------|----|---|
| RL3  | 13 | 1 |
| RL4  | 14 | 1 |
| RL5  | 15 | 1 |
| RR0  | 0  | 2 |
| RR2  | 2  | 2 |
| RR4  | 4  | 2 |
| RR6  | 6  | 2 |
| RR8  | 8  | 2 |
| RL10 | 10 | 2 |
| RL12 | 12 | 2 |
| RL14 | 14 | 2 |
| RL0  | 0  | 0 |
| RL1  | 1  | 0 |
| RL2  | 2  | 0 |
| RL3  | 3  | 0 |
| RL4  | 4  | 0 |
| RL5  | 5  | 0 |
| RL6  | 6  | 0 |
| RL7  | 7  | 0 |
| RL8  | 8  | 0 |
| RL9  | 9  | 0 |
| RL10 | 10 | 0 |
| RL11 | 11 | 0 |
| RL12 | 12 | 0 |
| RL13 | 13 | 0 |
| RL14 | 14 | 0 |
| RL15 | 15 | 0 |
| RL0  | 0  | 4 |
| RL4  | 4  | 4 |
| RL8  | 8  | 4 |
| RL12 | 12 | 4 |

\*\*\*\*\*)

```

545 procedure oneoperands(index: integer; var correct: boolean);
546
547
548 (* This procedure encodes the following types of opcodes
549    DI,EI,DAB,FXISB,EXISL,EXISL,REI,SC,NEG,NEGB,COM,COMB,
550    TEST,TESTB,TESTL,CLR,CIRB,CALL,LPDS,CALR *)
551
552
553 var
554   which: permit;
555   vi, nvi: boolean;
556   (* these parameters are used only in the encoding of the
557      the opcodes DI and EI that they do not follow the basic
558      method *)
559   token: one;
560   final, value1, value: real;
561
562   begin (*1*)
563     correct := true;
564     final := 0;
565     skipblanks(0);
566     case index of
567       167,169: (* DI , EI *)
568         begin (*2*)
569           gettoken (3,token,correct);
570           (* get the first token *)
571           nvi := false;
572           vi := false;
573           if correct then begin (*3*)
574             if token = 'NVI' then
575               (* it checks what kind of token it is
576                  because the tokens NVI and VI can be
577                  written in any order *)
578                 nvi := true
579                 else if token = 'VI' then

```

```

601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642

vi := true
else begin
    errormessage (28);
    correct := false ;
end;
end else
    errormessage (28);
    if correct then begin (*4*)
        (* the first token was correct *)
        skipblanks (0);
        if ch = ',' then
            begin(*5*)
                load;
                (* only if ',' is encountered it checks
                for other token because the opcode may
                has one or two operands *)
                skipblanks (0);
                gettoken (3,token,correct);
                if correct then begin (*6*)
                    if nvi then
                        correct := token = 'VI'
                    (* it checks that the second token is not
                    the same with the first one *)
                else
                    correct := token = 'NVI';
                    if not correct then
                        errormessage (28);
                    end;
                if correct then if token = 'VI' then
                    vi := true
                (* it sets the boolean variable which token has
                been found *)
            else
                nvi := true;
            end;

```

```

665 if correct then begin (*7*)
666   final := opcode[index].base;
667   if nvi then final := final +1;
668   (* calculates the final value *)
669   if vi then final := final +2;
670 end;
671 end;
672 end;
673 4, 5, 6, 7:
674 begin (* DAB, EXIS, EXTSB, EXTSL *) (*2*)
675   getoperand(type1, value, value1, which, correct);
676   (* all the above opcodes have the same type
677   of operand register *)
678   (* type1 is only req *)
679   if correct then begin (*3*)
680     (* includes all the opcode because there is only one
681       type of operand *)
682     case index of
683       (* this case checks if the registers are
684       compatible with the type of the opcode *)
685       4,6:
686         begin (*5*)
687           correct := value1 = 0
688         end; (*DAB, EXTSB *) (*5*)
689       5:
690         begin (*6*)
691           correct := value1 = 1
692         end; (* EXTSL *) (*6*)
693       7:
694         begin (*8*)
695           correct := value1 = 2
696         end
697       end
698     end (* EXTSL *) (*8*); (*case *)
699   if correct then
700     final := opcode[index].base + value * 16
701   end
702 end
703
704
705
706
707

```



```

785 xi:
786   final := x + value1 * 16
787   end; (*end case *)
788   case index of
789     10, 12, 14, 17, 92:
790       begin (* NEG, COM, TEST, TSET, CLR *)
791         (* it checks validity of the used register only
792         if the operand was register*)
793         if which = reg then
794           correct := value1 = 1
795         end;
796       11, 13, 15, 18, 93:
797         begin (* NEGB, COMB, TESTB, TSETB, CLRB *)
798           if which = reg then
799             correct := value1 = 0
800           end;
801       16:
802         begin (* TESTL *)
803           if which = reg then
804             correct := value1 = 2
805           end
806         end; (*end case *)
807       final := final + opcode(index).base;
808       (* calculates the final value of the opcode*)
809       if not correct then
810         errormessage(26)
811       end
812     end; (*20*)
813   19, 118:
814     begin (*30*) (* CALL, LPDS *)
815       getoperand(type4, value, value1, which, correct);
816       (* type4 are i, d, xi *)
817       if correct then begin
818         case which of
819           (* the final value of the opcode depends on the
820           A20
821           A21
822           A23
823           A24
824           A25
825           A26
826           A27

```





```
904  
905 end;  
906 if not correct and (index <> 167 ) and (index <> 168) then  
907     errormessage(28)  
908 end; (* oneoperands *)  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947
```

```

961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002

procedure twoop1(index: integer; var correct: boolean);
(* This procedure encodes the following types of
   opcodes:
   ADD,ADDB,ADUL,CP,CPB,CPL,DIV,DIVL,MULT,MULTL,
   SUB,SUBB,SUBL,AND,ANDB,OR,ORB,XOR,XORB,LD,LDB
   LDL,FX,EXB,SBC,SRCB,ADC,ADCB,ADK,LDK,LDAR,
   LDA,RLOB,RRDR *)

var
  which, which2: permit;
  relative : boolean;
(* is used by the opcode LDAR for the case of relative
   address of type 3+data *)
  final, value1, value: real;
  value2, value3: real;
  type21, type20: current;
  an: integer;
(* This variable is used to define if the data are
   IM and byte because the byte data use only the
   half upper byte of a word
   Also it is used denoting that the data are long word
   in the cases of the opcodes ADDL, SUBL,MULT,DIVL R, IM *)
  special: boolean;
(* This variable is used only for the opcode CP and CPB to
   distinguish the two different permitted combinations of
   operands *)

begin (*1*)
  special := false;
  relative := false;
  an := 4;
  final := 0;
  skipblanks(0);

```

```

1025 case index of
1026   21, 22, 23, 24, 25, 26, 27,
1027   28, 29, 30, 31, 32, 33, 34,
1028   35, 36, 37, 38, 39, 40, 41,
1029   42, 43, 44:
1030   begin (*2*)
1031     (*ADD, ADDR, ADDL, CP, CPB, CPL, DIV, DIVL, MULT, MULIL,
1032     SUB, SUBR, SUBL, AND, ANDR, OR, ORB, XOR, XORR, LB, LDB,
1033     LPL, EX, FXR *)
1034     if (index = 24) or (index = 25) then
1035       type21 := type3 (* req, ir, da, xi case of opcode CP, CPB*)
1036     else
1037       (* The opcode CP, CPB has two permitted combinations
1038       of operands :
1039       CP (B) R, I req, im, ir, da, x ]
1040       or
1041       CP (B) ldst, da, x ] , im
1042       So if the first operand is req. then it has the
1043       first combination *)
1044       type21 := type1;
1045       (* All the rest types of opcodes *)
1046       getoperand(type21, value, value1, which, correct);
1047       (*type1 = req, value = index of register,
1048       value1 = length of register*)
1049       skipblanks(1);
1050       (* it skips the blanks till to encountered an
1051       semicolon *)
1052       load;
1053       if correct then begin (*3*)
1054         skipblanks(0);
1055         (* skips the rest of the blanks *)
1056         if ((index = 24) or (index = 25)) and (which in lir, da, xi))
1057           (* the second operand of the CPR and CP opcode depends
1058           on the type of the type of the first one *)
1059           then begin
1060
1061
1062
1063
1064
1065
1066
1067

```

```

1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121

type20 := [im];(* case of CP P or CPB ir or da or xi,im*)
special := true
end else if (index = 43) or (index = 44) then
  type20 := type3 (* case of FX,EXD*)
else
  type20 := type8;
  getoperand(type20, value2, value3, which2, correct);
  (* type8 = req,im,ir,da,xi
  value2 = address,value of index of register,
  value3 = xregister,or length*)
  if correct then begin (*4*)
    case which2 of
      (* the addressing mode depends on the second
      operand and this affect the base value
      of the opcode *)
      req:   final := value2 * 16 + R;
      ir:    final := value2 * 16 + IR;
      da:    final := DA;
      xi:    final := X + value3 * 16;
      im:
        begin
          if (index = 23 ) or (index = 33)
            or (index = 42 ) then
            an := 8;
            (* ADDL , SUBL, LDL R, JM *)
            final := IM
          end;
        end;
      end; (* case *)
      final := final + value;
      (* the following case statement checks the
      validity of the registers of the first

```

```

1144 operand and of the second if it exists*)
1145 case index of
1146 21, 24, 31, 34, 36, 38, 40,
1147 43:
1148
1149   begin (*5*)
1150     (* word opcodes *)
1151     if which = reg then
1152       correct := value1 = 1;
1153     if which2 = reg then
1154       correct := value3 = 1
1155     end; (*5*)
1156 22, 25, 32, 35, 37, 39, 41,
1157 44:
1158
1159   begin (*6*) (* byte type opcodes *)
1160     if which = reg then
1161       correct := value1 = 0;
1162     if which2 = reg then
1163       correct := value3 = 0;
1164     an := 2
1165   end; (*6*)
1166 23, 26, 33, 42:
1167
1168   begin (*7*) (* long words *)
1169     correct := value1 = 2;
1170     if which2 = reg then
1171       correct := value3 = 2
1172     end; (*7*)
1173 29, 27:
1174
1175   begin (*8*)
1176     (* this is a special case the MULT
1177       and DIV opcodes use for the destination
1178       operand long register and for the source
1179       normal one word register *)
1180     (* MULT, DIV*)
1181     correct := value1 = 2;
1182     if which2 = reg then
1183
1184
1185
1186
1187

```

```

1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241

```

```

        correct := value3 = 1
    end; (* 8 *)
30, 28:
    begin (* 9 *)
        if which2 = im then
            an := 8;
            (* MUL1, DIVL *)
            (* the MUL1L and DIVL use for the destination
               quadruples registers and for the source
               long registers *)
            correct := value1 = 4;
            if which2 = reg then
                correct := value3 = 2
            end
        end (* 9 *)
    end (* case *)
    if correct then begin
        if special then begin
            (* then it means CP [ir, da, x], im
               In the table are saved only the
               base values of the opcode for the
               other combination of operands *)
            if index = 24 then
                final := 3329
            end (* CP *)
        else
            final := 3073;
        end (* CP8 *)
        if which = ir then
            final := final + value * 16
        end (* also the calculations formats are
               different between the two combina
               nations modes *)
        else if which = xi then
            (* x address *)
            final := final + value * 16
        end
    end

```

```

1265 else
1266     final := final + DA
1267     (* da address *)
1268 end else
1269     final := final + opcode[index].base
1270     (* normal mode of combination *)
1271 end else
1272     errormessage(26)
1273     (*4*)
1274     errormessage(28)
1275     (*3*)
1276     errormessage(28)
1277 end; (*2*)
1278
1279
1280
1281
1282
1283 begin
1284     (*SRC,SRCB,ADC,AUCB,LDK,LDAR,LDA,RLDB,RPDH*)
1285     (*20*)
1286
1287     getoperand(tyoe1, value, value1, which, correct);
1288     (*type1 = req,value= index of reg,
1289     value1 = length of req *)
1290     skipblanks(1);
1291     (* skips to find the ', ' *)
1292     load;
1293     if correct then begin (* 21*)
1294         skipblanks(0);
1295         (* this case statement selects
1296         the correct mode of subset of ope-
1297         rands *)
1298         if index = 51 then
1299             type20 := type9 (*LDA req, [da,xi]*)
1300
1301
1302
1303
1304
1305
1306
1307

```



```

1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
else if index = 50 then begin
  type20 := type7; (* LDAR req, ra*)
  if ch = 'S' then
    relative := true;
  end
else if index = 49 then
  type20 := type6 (* LDK req, im *)
else
  type20 := type1; (* R *)
  (* SBC,SBCB,ADC,ADCR,RLDB,RRDB R,R *)
  getoperand(type20, value2, value3, which2, correct);
  (* value2 = address, or value or index of the reg
    value3 := x register*)
  if correct then begin (* 22*)
    case index of
      (* checks validity of the registers *)
      45, 47:
        correct := (value1 = 1) and (value3 = 1);
      (* SBC,ADC *)
      46, 48, 52, 53:
        correct := (value1 = 0) and (value3 = 0);
      (* SBCB,ADCB,RLDB,RRDB *)
      49, 50, 51:
        correct := value1 = 1
    end (* LDK*); (*cuse*)
    if correct then begin (* 23*)
      final := opcode[index].base;
      (* it calculates each final value
        depending on the type of the opcode*)
      case index of
        45, 46, 47, 48, 52, 53:
          (* SBC,SBCB,ADC,ADCB,RLDB,RRDB *)
          final := final + value + value2 * 16;
        49:
          final := final + value * 16 + trunc(value2) mod 16;

```

```

1384 (* LDK *)
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
50:
begin
  final := final + value; (* LDA*)
  if relative then
    value2 := value2 - 2;
  end;
51:
begin (* LDA*)
  final := final + value;
  if which2 = xi then
    final := final + value3 * 16
  end
  end (* LDA*);
  end else (*case*) (*23*)
    errormessage(20)
  end else (*22*)
    errormessage(28)
  end else (*21*)
    errormessage(28)
  end
end (*20*); (* case *)
setmem(PC, final, 4);
PC := PC + 2;
(* sets the first word of the opcode *)
if (which2 in lda, xi, im, ral) and (index <> 49) or special
and (which <> ir) then begin
  if not special then begin
    (* all the other cases *)
    setmem(PC, value2, an);
    PC := PC + 2;
    if an = d then
      PC := PC + 2
    end else begin
      (* special case of CP (R) (ir,da,x) , im *)

```

```

1441
1442
1443
1444
1445
1446     if which = ir then
1447         value := value2;
1448     (* so only im data no address *)
1449     setmem(PC, value, 4);
1450     PC := PC + 2;
1451     if which <> ir then begin
1452     (* so im data after the address *)
1453         setmem(PC, value2, an);
1454         PC := PC + 2;
1455     end
1456     end;
1457 end
1458 end; (*1*)
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481

```

```

1504 procedure twoop2(index: integer; var correct: boolean);
1505
1506 (* this procedure encodes the following types of opcodes
1507    RLB,RL,RLCB,RLC,RRB,RR,RRCB,RRC,SDA,SDAL,SDLB,
1508    SLL,DJNZ,DRJNZ,SLAR,SLA,SLAL,SLL,SILL,SKAB,SRA,SRAL
1509    SRLB,SRL,SKLL *)
1510
1511 var
1512 done: boolean;
1513 (* this variable is used to denote that it does not
1514    exist second operand *)
1515 which, which2: permit;
1516 final, value1, value: real;
1517 value2, value3: real;
1518 type20: current;
1519
1520 begin (*1*)
1521   which := nil;
1522   which2 := nil;
1523   done := false;
1524   value2 := 0;
1525   correct := true;
1526   final := 0;
1527   skipblanks(0);
1528   getoperand(typel, value, value1, which, correct);
1529   (* the first operand is always register type for
1530      all the types of uncodes *)
1531   skipblanks(0);
1532   (* it does not use the procedure skipblanks (1) because
1533      the opcodes RLB till RRC may dont have second operand
1534      in the case that the shift will be executed only once*)
1535   if (ch <> last) and (ch <> '!') then
1536     load;

```

```

1561
1562
1563
1564
1565 if (index < 95) and (index < 96) then begin (* 1a *)
1566 (* all the cases except the case of the opcodes
1567   DJNZ and DJNZ *)
1568   if correct then begin (* 3a *)
1569     skipblanks(0);
1570     if (index < 62) or (index > 96) then
1571       type20 := lim1(* RLB,RL,KLC,RRB,RR,RRCB,KRC,RRCB,RRRC*)
1572       (*SLAB,SLA,SLAL,SLLB,SLI,SLLL,SRAB,SKA,SKAL,SRLB,SRL,SRLL *)
1573     else
1574       type20 := lreq; (*SDAB,SDA,SDAL,SDLB,SDL,SDLL *)
1575     if (ch <> last) and (ch <> '!') then
1576       (* it checks if it exists second operand *)
1577       getoperand(type20, value2, value3, which2, correct)
1578     else
1579       begin
1580         done := true;
1581         value2:=0;
1582         (* means that it does not exist second operand
1583           so the default value is 0 *)
1584       end;
1585   if correct then begin (* 4a *)
1586     (* checks if the no existence of second operand
1587       is permitted *)
1588     if index < 62 then
1589       (* RLB till RRC *)
1590       correct := (value2 = 2) or (done and (value2 = 0))
1591     else if index < 68 then
1592       correct := (value2 < 16) and not done
1593     (* the maximum value that is permitted is 15 *)
1594     (* SDAB .. SDLL *)
1595     (* the second operand is mandatory *)
1596     else if index > 96 then
1597       correct := not done;
1598     (* always is needed a second operand for the
1599       opcodes SLAB .. SRLL*)
1600
1601
1602

```

1624  
1625  
1626  
1627  
1628  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649  
1650  
1651  
1652  
1653  
1654  
1655  
1656  
1657  
1658  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667

```
if correct then begin (* 5*)
  final := opcode[index1.base + value * 16;
  if index < 62 then
    (* opcodes RLB .. RRC *)
    final := final + value2;
  case index of
    (* checks for validity of the registers *)
    54, 56, 58, 60, 62, 65, 97,
    100, 103, 106:
      (* bytes opcodes *)
      correct := value1 = 0;
    55, 57, 59, 61, 63, 66, 98,
    101, 104, 107:
      (* words opcodes *)
      correct := value1 = 1;
    64, 67, 69, 102, 105, 108:
      (* long words opcodes *)
      correct := value1 = 2
  end;
  if index > 102 then (* SRAR, SRA, SRAL, SRLR, SKL, SKLL *)
    (* changes the values in negative because the
       CPU by the positive or negative value knows
       if the shift is left or right *)
    (* changes the number to
       negative one *)
    value2 := 65536 - value2;
  if not correct then
    errormessage(26)
  end else
    errormessage(23)
  end else
    errormessage(28)
  end else
    errormessage(28)
  end else begin (* 1a*) (*1b*)
```

```

1681 (* DJNZ,DBJNZ *)
1682 if correct then begin (* 1c*)
1683   skipblanks(0);
1684   getoperand(type7, value2, value3, which2, correct);
1685   (* type7 is relative address *)
1686   if index = 95 then
1687     (* DJNZ *)
1688       correct := value = 1
1689       (* checks for validity of register *)
1690   else
1691     correct := value = 0;
1692     (* DBJNZ *)
1693     if not correct then
1694       errormessage(26);
1695     final:=opcode[index1].base+value * 256+trunc(value2) mod 128
1696     (* it performs mod operation because the maximum
1697       permitted address is +- 127
1698       *)
1699   end else (* 1c*)
1700     errormessage(28)
1701   end; (* 1ha*)
1702   setmem(PC, final, 4);
1703   PC := PC + 2;
1704   if (index > 61) and (index <> 95) and (index <> 96) then begin
1705     setmem(PC, value2, 4);
1706     PC := PC + 2
1707   end
1708   end; (* twoop2 *)
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722

```



```

1803 procedure twoop3(index: integer; var correct: boolean);
1804
1805
1806
1807 (* This procedure encodes the following types of opcodes :
1808
1809     DFCB,DEC,INCB,INC,BITR,BIT,PESB,RES,SFIB,SET,STORE,
1810     STOREB,STOREL,POP,POPL,PUSH,PUSHL,JP,JR,ICC,ICLB
1811     LDRB,LDR,LURL *)
1812
1813 var
1814     which, which3, which2: permit;
1815     relative : boolean;
1816     (* this variable is used by the opcode LDR and
1817        LURL in the case of relative address of type
1818        $+data *)
1819     fin, final, value1, value: real;
1820     value2, value3: real;
1821     type21, type20: current;
1822     an: integer;
1823     (* this variable is used to denote if the data are
1824        im and byte *)
1825     done: boolean;
1826     (* this variable denotes that it does not exist
1827        second operand *)
1828
1829
1830
1831
1832
1833     begin (*1*)
1834         done := false;
1835         relative := false;
1836         an := 4;
1837         which := nil;
1838         which2 := nil;
1839         value2 := 0;
1840
1841
1842
1843
1844
1845
1846
1847

```



```

1924 if not (ch in ['!', last]) then
1925   getoperand(type20, value2, value3, which2, correct)
1926   (* gets the second type of operand if it is exist*)
1927 else
1928   begin
1929     value2 := 0;
1930     done := true;
1931     (* means that the second operand does not exist *)
1932   end;
1933   if correct then begin (*4*)
1934     if index > 71 then
1935       correct := not done;
1936       (* in the opcodes of type BIT the second
1937         operand is obligatory and so it checks
1938         if it was existed *)
1939       (* because im is obligatory *)
1940       if correct then begin (*5*)
1941         case which of
1942           (* the base value of the opcode is
1943             affected by the address mode of the
1944             first operand *)
1945           req:
1946             final := p + value * 16 + value2;
1947           ir:
1948             final := IR + value * 16 + value2;
1949           da:
1950             final := DA + value2;
1951           xi:
1952             final := x + value * 16 + value2
1953         end;
1954       if (which = req) or (which2 = req) then
1955         (* if the first or the second operands where
1956           registers it checks for their validity *)
1957         case index of
1958           08, 10, 72, 74, 76:

```

```

1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021

begin
(* DECb,INCh,RIb,RFSb,SEb *)
  if which = reg then
    (* byte opcodes *)
    correct := value1 = 0;
    if which2 = reg then
      correct := correct and (value3 = 0)
    end;
  69, 71, 73, 75, 77:
begin
(* DEC,INC,RI,RES,SET *)
(* word opcodes *)
  if which = reg then
    correct := value1 = 1;
    if which2 = reg then
      correct := correct and (value3 = 1);
    end
  end;
  (* case *)
  if not correct then
    errormessage(26);
  case index of
    (* the evaluation of the final value depends
       on the type of the opcode *)
    68, 69, 70, 71: (*DECR,DEC,INCR,INC *)
      begin
        final := final + opcode[index].base - 1;
        if value2 = 0 then
          final := final + 1;
        (* because the 0 denotes one decrement *)
      end;
    72, 73, 74, 75, 76, 77:
      begin
        (*BITB,RI,RESb,RES,SFIb,SET *)
        (* there are two type of combinations of operands
           either

```

```

2045 BIT R,R
2046 or
2047   BIT (ir,dax,reg), im *)
2048   fin := opcode[index].base;
2049   if which2 = reg then begin
2050     (* so BIT R,R *)
2051     final := fin + value2;
2052     value := value * 256
2053   end else
2054     final := final + fin
2055     (* so type BIT (dax,ir,reg), im *)
2056   end
2057 end
2058 end else (* case of index of *) (* 5*)
2059   errormessage(23)
2060   end else (* 4*)
2061     errormessage(28)
2062   end else
2063     errormessage(28)
2064   end; (*2*)
2065
2066 78, 79, 80, 81, 82, 83, 84,
2067 85, 86, 87, 88, 89, 90, 91:
2068 begin (*10*)
2069
2070   (*STORE,STORER,STOREL,POP,POPL,PUSH,
2071   PUSHL,JP,JR,ICC,ICCB,LDRB,LDR,LDRL *)
2072   if index > 88 then (* LDRB,LDR,LDRL *)
2073     (* selects the first type of operand *)
2074     type20 := [reg, ral]
2075     else if index > 84 then (* JP,JR,ICCB,ICC *)
2076       type20 := lcc
2077     else if index > 82 then (* PUSH,PUSHL *)
2078       type20 := lirl
2079     else if index > 80 then (* POP,POPL *)

```

```

2101 type20 := [req, ir, da, xil]
2102 else
2103     (*STORE,STORE,STOREL *)
2104     type20 := [ir, da, xil];
2105     if ch = 'f' then
2106         relative := true;
2107         getoperand(type20, value, value1, which, correct);
2108         skipblanks(1);
2109         (* the second operand is mandatory *)
2110         load;
2111         if correct then begin (*11*)
2112             skipblanks(0);
2113             (* the second type of operand depends on the opcode*)
2114             if ch = 'f' then
2115                 relative := true;
2116             case index of
2117                 89, 90, 91:
2118                     type21 := [ra, req]; (* LDRB, LDR,LDRL *)
2119                 87, 88:
2120                     type21 := [req]; (* ICCB,ICCA*)
2121                 86:
2122                     type21 := [ra]; (* JR*)
2123                 85:
2124                     type21 := [ir, da, xil]; (* JP *)
2125                 83, 84:
2126                     type21 := [req, im, da, xil]; (* PUSH,PUSHL *)
2127                 81, 82:
2128                     type21 := [ir]; (* POP, POPL *)
2129                 78, 79, 80:
2130                     (*STORE,STORE,STOREL *)
2131                     type21 := [req, im]
2132             end (* STORE,STOREB,STOREL *) (* case index of *)
2133             getoperand(type21, value2, value3, which2, correct);
2134             (* gets the second operand *)
2135             if correct then begin (*12*)
2136                 (* the JR,PUSH and PUSHL opcodes change base
2137                     value depending on the second operand

```



```

2164 for this reason it is used a dummy variable
2165 for the search the which3 *)
2166 if (index = 85) or (index = 83 ) or (index = 84) then
2167   (*JR,PUSH,PUSHL, *)
2168   which3 := which2
2169 else
2170   which3 := which;
2171 case which3 of
2172   reg:
2173     final := R;
2174   ir,cc,ra,im:
2175     final := IR;
2176   da,xi:
2177     final := RA;
2178 end; (* case of which3 of *)
2179 final := final + opcode(index).base;
2180 case index of
2181   (* the final calculation of the value depends
2182     on each type of opcode *)
2183     91, 90, 89:
2184       begin (* LDRB,LDR,LDRL *)
2185         (* there are permitted two types of
2186           operands either LDR R,REL or
2187             LDR REL, R *)
2188         if (which = reg) and (which2 = ra) then
2189           begin
2190             final := final + value - R;
2191             if relative then
2192               value2 := value2 -2;
2193             end
2194           else if (which = ra) and (which = reg) then
2195             begin
2196               final := final + value2 + 512;
2197               if relative then
2198                 value := value -2
2199             end
2200           end
2201         end
2202       end
2203     end
2204   end
2205   end
2206   end
2207   end

```



34

```

2284 (* PUSH LR, IM *)
2285 final := final + value * 16
2286
2287 end;
2288
2289 81, 82:
2290 begin (* POP, POPL *)
2291   if not (which in [xi,da] ) then
2292     (* so ir or reg *)
2293     final := final + value + value2 * 16
2294   else
2295     if which = xi then
2296       final := final + value1 + value2 * 16
2297     else
2298       (* so da *)
2299       final := final + value2 * 16;
2300     end;
2301   78, 79, 80:
2302   begin (* STORFB, STORE, STORTL *)
2303     if index = 79 then (* STOREB *)
2304       an := 2;
2305       if which2 = im then begin
2306         (* there are two types of encoding
2307            either
2308            STORE (ir,da,x) , reg
2309            or
2310            STORE (ir,da,x) , im
2311            *)
2312         final := final - opcode(index).base;
2313         if index = 78 then (* STOREL *)
2314           final := final + 3328 + 5
2315         (* the im type encoding is not in the
2316            table and so it is provided in the
2317            algorithm *)
2318       else
2319         final := final + 3072 + 5
2320       end else begin
2321
2322
2323
2324
2325
2326
2327

```



```

2404 correct := value3 = 1;
2405 (*LDR,IRC,PUSH,STORE *)
2406 91, 84, 80;
2407
2408 correct := value3 = 2;
2409 (*LDRL,PUSHL,STOREL *)
2410 81, 82, 85, 86;
2411 null
2412
2413 end;
2414 if not correct then
2415   errormessage(20)
2416   end else
2417     (* case *) (*12*)
2418     errormessage(23)
2419     end else
2420       (*11*)
2421       errormessage(28)
2422
2423 end
2424 end (*10*); (* case big *)
2425 setmem(PC, final, 4);
2426 (* sets the memory to the result of the first word*)
2427 PC := PC + 2;
2428 if index < 83 then begin
2429   if (which = xi) or (which = da) then begin
2430     setmem(PC, value, 4);
2431     PC := PC + 2
2432   end
2433 end else begin
2434   if (which? in [xi, da, ra] ) and (index <> 86) then begin
2435     setmem(PC, value2, 4);
2436     PC := PC + 2
2437   end
2438 end;
2439 if ((index = 78) or (index = 79)) and (which2 = im) then begin
2440   (* only in the case of STORE IM the opcode consists
2441     from three words *)
2442   setmem(PC, value2, an);
2443   PC := PC + 2
2444
2445
2446
2447

```

2461  
2462  
2463  
2464  
2465  
2466  
2467  
2468  
2469  
2470  
2471  
2472  
2473  
2474  
2475  
2476  
2477  
2478  
2479  
2480  
2481  
2482  
2483  
2484  
2485  
2486  
2487  
2488  
2489  
2490  
2491  
2492  
2493  
2494  
2495  
2496  
2497  
2498  
2499  
2500  
2501  
2502

end  
end: { twoons }

```

2524 procedure twoop4(index: integer; var correct: boolean);
2525
2526
2527
2528 (* This procedure encodes the following opcodes :
2529    COMFLG,RESFLG,SFIFLG,INR,IN,OUTR,OUT,LDCILB,LDCIL *
2530
2531
2532 var
2533   which, which2: permit;
2534   fin, final, value1, value: real;
2535   value2, value3: real;
2536   type21, type20: current;
2537   kilo: set of char;
2538   (* this set of characters is used to check
2539      if the flags in the opcodes of type COMFLG
2540      are valid and if they are used more than one
2541      time *)
2542
2543
2544
2545 begin
2546   kilo := {'C', 'Z', 'S', 'P', 'V'};
2547   which := nil;
2548   which2 := nil;
2549   value2 := 0;
2550   correct := true;
2551   final := 0;
2552   skipblanks(0);
2553   case index of
2554
2555
2556
2557     115, 116, 117:
2558       begin (*1*)
2559         (*COMFLG,RESFLG,SFIFLG*)
2560         repeat
2561
2562
2563
2564
2565
2566
2567

```

500



```

2644 load;
2645 if ch = ',' then begin
2646   (* yes exists other flag *)
2647   load;
2648   skipblanks(0)
2649 end
2650 until (ch in ['!', last]) or not correct;
2651 final := final * 16 + opcode[index].base
2652 end; (*2*)
2653 109, 110, 111, 112, 113, 114:
2654 begin
2655   (* INB,IN,OUT,LDCTLB,LDCTL*) (* 5*)
2656   case index of
2657     109, 110:
2658       type20 := [req]; (* INB,IN*)
2659       (* selects the type of the first operand
2660        depending on the type of the opcode *)
2661     111, 112:
2662       type20 := [ir, dal]; (*OUTB,OUT*)
2663     113, 114:
2664       type20 := [control, reg]
2665     end (* LDCTLB,LDCTL*); (* case*)
2666     getoperand(type20, value, value1, which, correct);
2667     skipblanks(1);
2668     load;
2669     if correct then begin (*b*)
2670       skipblanks(0);
2671       case index of
2672         109, 110:
2673           (* selects the second type of operand depending
2674            on the type of opcode *)
2675         type21 := [ir, dal];
2676         111, 112:
2677           type21 := [req];
2678         113, 114:
2679
2680
2681
2682
2683
2684
2685
2686
2687

```

```

2701 begin
2702   if which = control then
2703     type21 := [req]
2704     (* there are two combinations either
2705       control, req or req,control *)
2706   else
2707     type21 := [control]
2708   end
2709   end; (*case*)
2710   getoperand(type21, value2, value3, which2, correct);
2711   if correct then begin (*/*)
2712     case index of
2713       109, 110, 111, 112:
2714         (* INR,IN,OUTR,OUT *)
2715         (* depending on the type of the opcode
2716           calculates the final value *)
2717         begin (*8*)
2718           final := opcode[index].base;
2719           if (which = req) and (which2 = ir) then
2720             final := final + value + value2 * 16
2721             (* IN (R) R, IR *)
2722           else if (which = req) and (which2 = da) then
2723             final := final + value * 16 + 4
2724             (* IN (R) R, DA *)
2725           else if (which = ir) and (which2 = req) then
2726             final := final + value * 16 + value2
2727             (* OUT (R) IR,R *)
2728           else
2729             final := final + 6 + value2 * 16;
2730             (*UII (R) DA, R*)
2731           case index of
2732             (* checks for validity of the registers *)
2733             109:
2734               (* INR *)
2735               correct := value1 = 0;

```

2764  
2765  
2766  
2767  
2768  
2769  
2770  
2771  
2772  
2773  
2774  
2775  
2776  
2777  
2778  
2779  
2780  
2781  
2782  
2783  
2784  
2785  
2786  
2787  
2788  
2789  
2790  
2791  
2792  
2793  
2794  
2795  
2796  
2797  
2798  
2799  
2800  
2801  
2802  
2803  
2804  
2805  
2806  
2807

```
110:
(* IN *)
correct := value1 = 1;
111:
(* OUTB *)
correct := value3 = 0;
112:
(* OUT *)
correct := value3 = 1
end;
if not correct then
  errormessage(26)
end; (*A8*)

113, 114:
begin (* 10*)
(* LDCILB , LDCIL *)
final := opcode[index1].base;
if which = reg then
  final := final + value * 16
(* if the first operand is register
  so the second is control *)
(* R , CONTROL *)
else
  final := final + value2 * 16;
(* so the first is control and the
  second is register *)
(* CONTROL , R *)
if which = control then
  fin := value + 9
else
  fin := value2 + 1;
final := final + fin;
```

```

2821 if which = reg then begin
2822   if index = 113 then
2823     correct := value1 = 0
2824     (* LDCILB so byte register *)
2825   else
2826     correct := value1 = 1
2827     (* so LDCIL so word register *)
2828   end else begin
2829     if index = 113 then
2830       correct := value3 = 0
2831       (* case LDCILB and register second operand *)
2832       (* the register is second operand
2833         so it checks the value3 *)
2834     else
2835       correct := value3 = 1
2836       (* case LDCIL and register second operand *)
2837     end;
2838   if not correct then
2839     errormessage(26)
2840   end
2841   end (* 10*)
2842   end else (* case*) (* 7*)
2843     errormessage(26)
2844   end else (* 6*)
2845     errormessage(28)
2846   end
2847   end (* 5*); (*case*)
2848   setmem(PC, final, 4);
2849   PC := PC + 2;
2850   if which = da then begin
2851     setmem(PC, value, 4);
2852     PC := PC + 2
2853   end;
2854   if which2 = da then begin
2855     setmem(PC, value2, 4);
2856   end;
2857   if which2 = da then begin
2858     setmem(PC, value2, 4);
2859   end;
2860   if which2 = da then begin
2861     setmem(PC, value2, 4);

```

```
2884      PC := PC + 2  
2885  
2886      end  
2887      end; { twond4 }  
2888  
2889  
2890  
2891  
2892  
2893  
2894  
2895  
2896  
2897  
2898  
2899  
2900  
2901  
2902  
2903  
2904  
2905  
2906  
2907  
2908  
2909  
2910  
2911  
2912  
2913  
2914  
2915  
2916  
2917  
2918  
2919  
2920  
2921  
2922  
2923  
2924  
2925  
2926  
2927
```

```

2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999

procedure threeoperands(index: integer; var correct: boolean);
(* This procedure encodes all the opcodes with three
   operands as follows :
       INDR,IND,INDRR,INDR,INIB,INI,INIRB,INIR
       OUTDB,OUTD,OUTIB,OUTI,OUTDRB,UTDR,OUTIRB,OTIK
       LDRB,LDD,LDDRB,LDDR,LDIR,LDI,LDIRB,LDIR
       LDM
       IRDB,IRDRB,IRIB,IRIRB,IRIDB,IRIDRB,IRIIR,IRIIRB
       *)
var
which, which2, which3: permit;
value, value1, value2, value3: real;
value4, value5: real;
second, third, final: real;
typed0 : current;

begin (* 1*)
  which := nil;
  which2 := nil;
  which3 := nil;
  value2 := 0;
  value4 := 0;
  value5 := 0;
  final := 0;
  skipblanks(0);
  if (index <> 94 ) then begin (* means no LDM type opcode*)
    (* all the opcodes except the LDM have the same algorithm*)
    getoperand(fir), value, value1, which, correct);
    (* the first operand is IP *)
    skipblanks(1);
    load;
    if correct then begin (*2*)

```

```

3004 skipblanks(0);
3005 (* correct so continue for the second operand *)
3006 getoperand(fir), value2, value3, which2, correct);
3007 skipblanks(1);
3008 (* and the second is IR *)
3009 load;
3010 if correct then begin (*3*)
3011 skipblanks(0);
3012 (* so correct continues for the third one *)
3013 getoperand(freq), value4, value5, which3, correct);
3014 if correct then begin (*4*)
3015   final := opcode[index].base + value2 * 10;
3016   (* calculates the final bvalue *)
3017   case index of
3018     119, 120, 123, 124, 127, 128, 129,
3019     130, 135, 136, 139, 140:
3020       second := 8;
3021     (* IN,UIF no repeat opcodes *)
3022     121, 122, 125, 126, 131, 132, 133,
3023     134, 137, 138, 141, 142, 143, 144,
3024     145, 146, 147, 149:
3025       second := 0;
3026     (* IRRR .. UIR..and LDR opcodes *)
3027     148, 150:
3028       (* IRIORB IRIKR *)
3029       second := 14
3030   end;
3031   second := second + value * 16 + value4 * 256;
3032   (* is the second word of the opcode *)
3033   if index < 143 then begin
3034     if odd(index) then
3035       (* checks for validity of the registers *)
3036       correct := value5 = 0
3037       (* bytes opcodes *)
3038     else
3039
3040
3041
3042
3043
3044
3045
3046
3047

```



```

3061
3062
3063
3064
3065
3066
3067
3068
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3100
3101
3102

```

```

        correct := values = 1
        (* words opcodes *)
    end else
        correct := values = 0;
        (* ALL CASES OF THE OPCODES OF TYPE TRDB*)
        correct := correct and ((values = 1) and (value1 = 1));
        (* checks that the ir registers were also word
           registers and only *)
        if not correct then
            errormessage(26)
        end else
            errormessage(28)
        end else
            errormessage(28)
        end else
            errormessage(28)
        end else
            errormessage(28);
        end (* of the others types of opcodes *)
        else begin (*3a*) (* means LDM type opcode *)
            getoperand(freg,ir,da,xl,value,value1,which,correct);
            (* the first operand may be all the above cases *)
            skipblanks(1);
            load;
        if correct then begin (* 31*)
            skipblanks(0);
            if which = reg then (* is load type LDM opcode*)
                (* so load and not store multiple *)
                type20 := (ir,da,xl)
                (* so the second type of operand must be one
                   of the types ir,da,xi *)
            else (* is store type LDM opcode*)
                type20 := (reg);
            getoperand (type20,value2,value3,which2,correct);
            skipblanks(1);
            load;
        if correct then begin (*31*)

```

```

3124 skipblanks(0);
3125 getoperand(fiml,value4,value5,which3,correct);
3126 (* the last operand and for the two cases is
3127 im *)
3128 value4 := trunc(value4) mod 16; (* max value 15*)
3129 (* the im value is not possible to be greater than
3130 15 *)
3131
3132 if correct then begin (* 33*)
3133   if which = req then begin (*39*)
3134     case which2 of
3135       ir : final := 1;
3136       (* depends on the address mode of the second
3137        operand the base value of the opcode is
3138        affected *)
3139       da,xi : final := DA + 1;
3140     end ;(*case*)
3141     final := final + opcode[index1].base;
3142     if which2 = ir then
3143       final := final + value2 * 16
3144     else if which2 = xi then
3145       final := final + value3 * 16;
3146     if which2 in [da,xi] then
3147       third := value2;
3148       second := value * 256 + value4;
3149       (* calculates the second word of the opcode*)
3150       end (*34*) (* load multiple*)
3151     else begin (*35*)
3152       (* in this the first operand affects the
3153        base value of the opcode *)
3154       case which of
3155         ir : final := 9;
3156         da,xi : final := DA + 9;
3157       end;
3158       final := final + opcode[index1].base;
3159       if which = ir then
3160
3161
3162
3163
3164
3165
3166
3167

```

```

3181         final := final + value * 16
3182     else if which = xi then
3183         final := final + value1 * 16;
3184     if which in lda,xil then
3185         third := value;
3186     second := value? * 256 + value4;
3187 end; (*35*)
3188 if which = reg then
3189 (* if the first operand was register it checks
3190 for validity *)
3191     correct := (value = 1)
3192 else
3193     correct := (value3 = 1);
3194 (* so is the second operand *)
3195 if which = ir then
3196     correct := correct and (value1 = 1);
3197 (* all the ir registers must be word registers*)
3198 if which? = ir then
3199     correct := correct and (value3 = 1);
3200 if not correct then
3201     errormessage(26);
3202 end (*33*)
3203 else
3204     errormessage(28);
3205 end (*32*)
3206 else
3207     errormessage(28);
3208 end (*31*)
3209 else
3210     errormessage(28);
3211 end (*30*)
3212 errormessage(28);
3213 end; (*30*)
3214 setmem(PC, final, 4);
3215 setmem(PC + 2, second, 4);
3216 PC := PC + 4;
3217 if (index = 94) and ((which in lda,xil) or (which? in lda,xil))
3218

```

```
then begin
  setmem (PL,third,4);
  PC := PC + 2;
end;
end; (*1*)
```

3243  
3244  
3245  
3246  
3247  
3248  
3249  
3250  
3251  
3252  
3253  
3254  
3255  
3256  
3257  
3258  
3259  
3260  
3261  
3262  
3263  
3264  
3265  
3266  
3267  
3268  
3269  
3270  
3271  
3272  
3273  
3274  
3275  
3276  
3277  
3278  
3279  
3280  
3281  
3282  
3283  
3284  
3285  
3286  
3287

```

3301
3302
3303
3304
3305
3306
3307
3308
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341

procedure fouroperands(index: integer; var correct: boolean);
(* this procedure encodes the opcodes with four operands
   as follows:
       CPDR.CPD,CPSDRB,CPDRB,CPIB,CPI,CPIRB,CPIR
       CPSDRB,CPSD,CPSDRB,CPSDR,CPSIB,CPSI,CPSIRB,CPSIR
       *)
var
    which, which2, which3, which4: permit;
    value, value1, value2, value3: real;
    value4, value5, value6, value7: real;
    second, final: real;
    type20: current;

begin (* 1*)
    which := nil;
    which2 := nil;
    which3 := nil;
    which4 := nil;
    value2 := 0;
    value4 := 0;
    value6 := 0;
    final := 0;
    correct := true;
    skipblanks(0);
    if index < 159 then
        type20 := lreal
    else
        type20 := lirl;
    (* less than 159 then CPDR .. CPIR else CPSDRB..CPSIR...*)
    getoperand(type20, value, value1, which, correct);
    (* gets the first operand *)

```

```

3364 if correct then begin (*2*)
3365   skipblanks(1);
3366   load;
3367   getoperand(fir1, value2, value3, which2, correct);
3368   (* get the second one which is always ir *)
3369   if correct then begin (*3*)
3370     skipblanks(1);
3371     load;
3372     getoperand(freg1, value4, value5, which3, correct);
3373     (* the third one is always register *)
3374     if correct then begin (*3a*)
3375       skipblanks(1);
3376       load;
3377       getoperand(fcc1, value6, value7, which4, correct);
3378       (* the fourth operand is condition code *)
3379       if correct then begin (*4*)
3380         final := opcodeindex1.base + value2 * 16;
3381         second := second + value * 10 + value4 * 256 + value6;
3382         (* calculates the second word of the opcode *)
3383         if index < 159 then begin
3384           if odd(index) then
3385             (* checks for register validity *)
3386             correct := value1 = 0
3387             (* bytes registers *)
3388           else
3389             correct := value1 = 1
3390             (* words registers *)
3391         end else
3392           correct := value1 = 1;
3393         correct := correct and ((value3 = 1) and (value5 = 1));
3394         (* checks for validity of the ir registers *)
3395         if not correct then
3396           errormessage(26)
3397             (*4*)
3398         end else
3399           errormessage(28)
3400         end
3401       end
3402     end
3403   end
3404 end
3405
3406
3407

```

```

3421
3422
3423
3424
3425
3426      end else (*3a*)
3427          errormessage(28)
3428              (*3*)
3429      end else
3430          errormessage(28)
3431              (*2*)
3432      end else
3433          errormessage(28);
3434      setmem(PC, final, 4);
3435      setmem(PC + 2, second, 4);
3436      PC := PC + 4;
3437      end;
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
3450
3451
3452
3453
3454
3455
3456
3457
3458
3459
3460
3461

```



```

3484 procedure updateLabels;
3485
3486
3487
3488 (* This procedure is used to update the positions of
3489 the memory where a label was referenced forward .So
3490 as the label is encountered the label table is up-
3491 dated and in the end of the encoding of all the op-
3492 codes is called to set all the forward references
3493 and also to check if a label was referenced without
3494 to be met.
3495 *)
3496
3497 var
3498 index,i: integer;
3499 an: real;
3500
3501 begin
3502   writeln (error);
3503   writeln (error, '      UPDATE THE FORWARD REFERENCED LABELS');
3504   for index := stalabel to labelindex do begin
3505     i := 0;
3506     while (labelist[index, i] <> -1) and (i <= maxlabel)
3507     (* checks if they exist memory locations in the asso-
3508      ciated list of the label *)
3509     do begin
3510       an := variable[index].initial;
3511       if an = -1 then begin
3512         errorcounter := errorcounter +1;
3513         (* means that there are but the label was not
3514          met *)
3515         writeln (error, ' ':4,'Label ',variable[index].name,
3516                 ' is referenced without to exist');
3517       end;
3518       if an <> -1 then
3519         setmem(labelist[index, i], an, 4);
3520
3521
3522
3523
3524
3525
3526
3527

```

```

3541
3542
3543
3544
3545
3546
3547
3548
3549
3550
3551
3552
3553
3554
3555
3556
3557
3558
3559
3560
3561
3562
3563
3564
3565
3566
3567
3568
3569
3570
3571
3572
3573
3574
3575
3576
3577
3578
3579
3580
3581
3582
3583
3584
3585
3586
3587
3588
3589
3590
3591
3592
3593
3594
3595
3596
3597
3598
3599
3600
3601
3602
3603
3604
3605
3606
3607
3608
3609
3610
3611
3612
3613
3614
3615
3616
3617
3618
3619
3620
3621
3622
3623
3624
3625
3626
3627
3628
3629
3630
3631
3632
3633
3634
3635
3636
3637
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
3650
3651
3652
3653
3654
3655
3656
3657
3658
3659
3660
3661
3662
3663
3664
3665
3666
3667
3668
3669
3670
3671
3672
3673
3674
3675
3676
3677
3678
3679
3680
3681
3682
3683
3684
3685
3686
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
3700
3701
3702
3703
3704
3705
3706
3707
3708
3709
3710
3711
3712
3713
3714
3715
3716
3717
3718
3719
3720
3721
3722
3723
3724
3725
3726
3727
3728
3729
3730
3731
3732
3733
3734
3735
3736
3737
3738
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749
3750
3751
3752
3753
3754
3755
3756
3757
3758
3759
3760
3761
3762
3763
3764
3765
3766
3767
3768
3769
3770
3771
3772
3773
3774
3775
3776
3777
3778
3779
3780
3781
3782
3783
3784
3785
3786
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799
3800
3801
3802
3803
3804
3805
3806
3807
3808
3809
3810
3811
3812
3813
3814
3815
3816
3817
3818
3819
3820
3821
3822
3823
3824
3825
3826
3827
3828
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
3850
3851
3852
3853
3854
3855
3856
3857
3858
3859
3860
3861
3862
3863
3864
3865
3866
3867
3868
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
3900
3901
3902
3903
3904
3905
3906
3907
3908
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
3950
3951
3952
3953
3954
3955
3956
3957
3958
3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
4000

```

```

3604 procedure intlabelopcode(token: one; var labop: boolean;
3605   var index: integer; var correct: boolean;
3606   var numoperands : integer );
3607
3608
3609 (* this procedure is used to check if the first
3610   encountered token in each line is a label or
3611   a opcode. So it searches first the list of the
3612   labels and then the list of the opcodes in the
3613   case of unsuccessful search in the list of the
3614   labels *)
3615
3616
3617 var
3618   lima, k: integer;
3619
3620 begin
3621   labop := false;
3622   k := maxvar;
3623   repeat
3624     k := k + 1
3625   until (token = variable[k].name) or (k >= labelindex);
3626   (* it starts from the max variable where the labels
3627     start till the maximum declared label *)
3628   if token = variable[k].name then begin
3629     labop := true;
3630     (* if it is label it sets the parameter labop *)
3631     correct := true;
3632     index := k
3633   end;
3634   if not labop then begin
3635     give (lima, token[0]);
3636     (* it returns the header of the list of the
3637       opcodes that have the first same character
3638       with the token *)
3639     repeat
3640
3641
3642
3643
3644
3645
3646
3647

```

```

3661
3662
3663
3664
3665
3666
3667
3668
3669
3670
3671
3672
3673
3674
3675
3676
3677
3678
3679
3680
3681
3682
3683
3684
3685
3686
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
3700
3701

(* so it continues to search for opcode *)
k := fast(lima);
lima := lima + 1;
until (token = opcode[k].name) or (fast(lima) = -1);
if token = opcode[k].name then begin
  labop := false;
  index := k;
  (* it reads all the contents of the opcode table *)
  numoperands := opcode[k].syntax;
  correct := true
end else
  correct := false
end
end
(* so the token is not either opcode so the search was
   unsuccessful *)
end; (* intlabelopcode *)

```

3724  
3725  
3726  
3727  
3728  
3729  
3730  
3731  
3732  
3733  
3734  
3735  
3736  
3737  
3738  
3739  
3740  
3741  
3742  
3743  
3744  
3745  
3746  
3747  
3748  
3749  
3750  
3751  
3752  
3753  
3754  
3755  
3756  
3757  
3758  
3759  
3760  
3761  
3762  
3763  
3764  
3765  
3766  
3767

```
procedure calopcode(var alldone: boolean);
```

```
(* This procedure is used after the scanning of the  
constants and the variables and it finds if the  
first token in each line is a label or a opcode.  
In the case of a label it checks if it follows  
the ':' symbol and if the rest of a line is empty.  
If the line is not empty then the following token  
must be an opcode and not a second label.If it  
follows a second label then it prints an error  
message.  
In the case of an opcode the procedure intlabeledopcode  
returns the index of the opcode in the table and  
also it returns the number of operands of the opcode  
and depending on these values it calls the appropriate  
procedure to continue the encoding of the opcode.  
If the first token was the reserved word 'END' it  
stops the search and it returns to the procedure op-  
code analyzer true with the parameter finished and so  
the search it stops *)
```

```
var  
done, correct, labon: boolean;  
(* done denotes that the line after the label is  
empty.  
correct denotes that the token is correct.  
labon denotes that the token is label if the  
value is true and opcode if the value is false*)  
numoperands, index: integer;  
(* numoperands denotes the number of operands of  
the opcode.  
index denotes the value of the index in the  
opcode table where the opcode was found *)  
token: one;
```

```

3781 value : real;
3782 (* this variable is used for the case of ORG statement
3783 to calculate the address *)
3784
3785
3786
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799
3800
3801
3802
3803
3804
3805
3806
3807
3808
3809
3810
3811
3812
3813
3814
3815
3816
3817
3818
3819
3820
3821

```

```

begin
  correct := true;
  done := false;
  skipcomments;
  gettoken(5, token, correct);
  (* gets the first token *)
  if (token <> 'END') and correct and (token <> 'ORG') (*2*)
  then begin
    (* if token equals to 'END' terminates the search *)
    intlabelopcode(token, labop, index, correct, numoperands);
    (* checks if the token is label or opcode *)
    if labop and correct then begin (*10*)
      (* yes it is label *)
      if variableindex.initial <> -1 then
        errormessage (39);
      variableindex.initial := PC;
      (* update the table of the labels with the
        address where the label was found and
        also it checks if the label was used before
        in different place *)
      skipblanks(0);
      if ch = ':' then begin (*11*)
        load;
        skipblanks(0);
        if (ch <> last) and (ch <> '!') then begin (*12*)
          (* checks if the line is empty *)
          gettoken(5, token, correct);
          (* else gets the second token *)
          if correct then begin (*13*)

```

```

3844 intlabelopcode(token, labop, index, correct, numoperands);
3845 (* checks what kind of token it is *)
3846 if labop then begin (*14*)
3847   (* it is not permitted a label to be followed
3848     by other label *)
3849   correct := false;
3850   errormessage(25)
3851   end
3852 end (*14*)
3853 end else begin      (*13*) (*12*)
3854   (* means that the line is empty or only
3855     comments *)
3856   clear;
3857   done := true
3858   end
3859 end else begin      (*11*)
3860   errormessage(24);
3861   load;
3862   if (ch <> last) and correct then
3863     skipblanks(2);
3864   (* checks what characters follow the token *)
3865   clear;
3866   correct := false
3867   end
3868 end; (*10*)
3869 if correct and not done then begin (*20*)
3870   case numoperands of
3871     (* selects the type of the procedure to encode
3872       the opcode depending on the number of operands
3873       and the index in the table *)
3874     0:
3875       zerooperands(index, correct);
3876     1:
3877       oneoperands(index, correct);
3878     2:

```



```

3901
3902
3903
3904
3905
3906
3907
3908
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941

```

```

    if index < 54 then
        twoop1(index, correct)
    else if index < 68 then
        twoop2(index, correct)
    else if index < 92 then
        twoop3(index, correct)
    else if index < 109 then
        twoop2(index, correct)
    else
        if index < 118 then
            twoop4 (index,correct);
        3:
        threepoperands(index, correct);
        4:
        fouroperands(index, correct)
    end;
    if ch <> last then
        skipblanks(?)
    end (*20*)
    else if not done then
        errormessage (18);
    clear
    end else begin (*2*) (*3*)
    (* else no correct or END or ORG *)
    if correct then begin (* 30 *)
        if token = 'END' then
            alldone := true
        else (* case ORG *)
            begin
                evalconst (value,correct,4);
                (* calculates the address *)
                if odd (trunc (value)) then
                    value := value +1;
                (* compensates for odd address *)
                if correct then begin

```

```

3964   if PC <= value then
3965       PC := trunc (value)
3966   else
3967       errormessage (36);
3968       (* means address less than the
3969         PC *)
3970       end; (* correct *)
3971       clear;
3972       end; (* ORG *)
3973       end (* correct*) else
3974       clear;
3975       end (* *)
3976       end; (* calopcode *)
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
4000
4001
4002
4003
4004
4005
4006
4007

```

```

4021 procedure opcodeanalyze(var finished: boolean);
4022 (* This procedure performs the following functions :
4023    - it initializes the subsets of the different
4024      types of operands combinations.
4025    - it calls repeatedly the procedure calopcode
4026      till the reserved word 'END' is encountered.
4027 *)
4028
4029 var
4030 correct, alldone: boolean;
4031
4032 begin
4033   type1 := fregl;
4034   type3 := freg, ir, da, xil;
4035   type4 := fir, da, xil;
4036   type5 := fccl;
4037   type6 := fim;
4038   type7 := fral;
4039   type8 := freg, im, ir, da, xil;
4040   type9 := fda, xil;
4041   alldone := false;
4042   skipcomments;
4043   (* skips empty lines and lines with only comments *)
4044   gettoken(5, token, correct);
4045   (* gets the first token that must be 'ENTRY' denoting
4046     that the opcodes follow *)
4047   if token = 'ENTRY' then begin
4048     clear;
4049     repeat
4050       skipcomments;
4051       calopcode(alldone)
4052     until alldone
4053   end;
4054   (* error so it terminates the whole operation *)

```

```
finished := all done
end; (* oncodeanalyze *)
```

4083  
4084  
4085  
4086  
4087  
4088  
4089  
4090  
4091  
4092  
4093  
4094  
4095  
4096  
4097  
4098  
4099  
4100  
4101  
4102  
4103  
4104  
4105  
4106  
4107  
4108  
4109  
4110  
4111  
4112  
4113  
4114  
4115  
4116  
4117  
4118  
4119  
4120  
4121  
4122  
4123  
4124  
4125  
4126  
4127

```

4141
4142
4143
4144
4145
4146
4147
4148
4149
4150
4151
4152
4153
4154
4155
4156
4157
4158
4159
4160
4161
4162
4163
4164
4165
4166
4167
4168
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
526

procedure openfile;
(* this procedure is called after the message and it
checks if the user wishes to quit else it reads the
user's filename and creates the object file name.*)

var
ch: char;
count: integer;

begin
count := 0;
(* it reads the user's input filename *)
repeat
read(ch);
if (ch = 'Q') and (count = 0) then
goto 100;
(* the user wishes to quit so it terminates the program*)
count := count + 1;
infile[count] := ch
until eoln(input) or (count = mfilename - 2);
errorfile := infile;
errorfile[count + 1] := '.';
(* creates the name of the errorfile *)
errorfile[count + 2] := 'n';
reset(source, infile);
(*opens the input source file *)
rewrite(error, errorfile);
(* creates the output file *)
rewrite(object, outfile);
writeln(error);
writeln(error);
writeln(error, ' ': 20, 'SOURCE FILE');
writeln(error, ' ': 16, '-----');
writeln(error);
writeln(error, ' :3, ' PC ', ' CODE');

```

```
writeln (error);  
end; (* openfile *)
```

4204  
4205  
4206  
4207  
4208  
4209  
4210  
4211  
4212  
4213  
4214  
4215  
4216  
4217  
4218  
4219  
4220  
4221  
4222  
4223  
4224  
4225  
4226  
4227  
4228  
4229  
4230  
4231  
4232  
4233  
4234  
4235  
4236  
4237  
4238  
4239  
4240  
4241  
4242  
4243  
4244  
4245  
4246  
4247

```

4261
4262
4263
4264
4265
4266
4267
4268
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
4300
4301
4302
4303
4304
4305
4306
4307
4308
4309
4310
4311
4312
4313
4314
4315
4316
4317
4318
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
4350
4351
4352
4353
4354
4355
4356
4357
4358
4359
4360
4361
4362
4363
4364
4365
4366
4367
4368
4369
4370
4371
4372
4373
4374
4375
4376
4377
4378
4379
4380
4381
4382
4383
4384
4385
4386
4387
4388
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
4400
4401
4402
4403
4404
4405
4406
4407
4408
4409
4410
4411
4412
4413
4414
4415
4416
4417
4418
4419
4420
4421
4422
4423
4424
4425
4426
4427
4428
4429
4430
4431
4432
4433
4434
4435
4436
4437
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
4450
4451
4452
4453
4454
4455
4456
4457
4458
4459
4460
4461
4462
4463
4464
4465
4466
4467
4468
4469
4470
4471
4472
4473
4474
4475
4476
4477
4478
4479
4480
4481
4482
4483
4484
4485
4486
4487
4488
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499
4500
4501
4502
4503
4504
4505
4506
4507
4508
4509
4510
4511
4512
4513
4514
4515
4516
4517
4518
4519
4520
4521
4522
4523
4524
4525
4526
4527
4528
4529
4530
4531
4532
4533
4534
4535
4536
4537
4538
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549
4550
4551
4552
4553
4554
4555
4556
4557
4558
4559
4560
4561
4562
4563
4564
4565
4566
4567
4568
4569
4570
4571
4572
4573
4574
4575
4576
4577
4578
4579
4580
4581
4582
4583
4584
4585
4586
4587
4588
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4599
4600
4601
4602
4603
4604
4605
4606
4607
4608
4609
4610
4611
4612
4613
4614
4615
4616
4617
4618
4619
4620
4621
4622
4623
4624
4625
4626
4627
4628
4629
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
4650
4651
4652
4653
4654
4655
4656
4657
4658
4659
4660
4661
4662
4663
4664
4665
4666
4667
4668
4669
4670
4671
4672
4673
4674
4675
4676
4677
4678
4679
4680
4681
4682
4683
4684
4685
4686
4687
4688
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4699
4700
4701
4702
4703
4704
4705
4706
4707
4708
4709
4710
4711
4712
4713
4714
4715
4716
4717
4718
4719
4720
4721
4722
4723
4724
4725
4726
4727
4728
4729
4730
4731
4732
4733
4734
4735
4736
4737
4738
4739
4740
4741
4742
4743
4744
4745
4746
4747
4748
4749
4750
4751
4752
4753
4754
4755
4756
4757
4758
4759
4760
4761
4762
4763
4764
4765
4766
4767
4768
4769
4770
4771
4772
4773
4774
4775
4776
4777
4778
4779
4780
4781
4782
4783
4784
4785
4786
4787
4788
4789
4790
4791
4792
4793
4794
4795
4796
4797
4798
4799
4800
4801
4802
4803
4804
4805
4806
4807
4808
4809
4810
4811
4812
4813
4814
4815
4816
4817
4818
4819
4820
4821
4822
4823
4824
4825
4826
4827
4828
4829
4830
4831
4832
4833
4834
4835
4836
4837
4838
4839
4840
4841
4842
4843
4844
4845
4846
4847
4848
4849
4850
4851
4852
4853
4854
4855
4856
4857
4858
4859
4860
4861
4862
4863
4864
4865
4866
4867
4868
4869
4870
4871
4872
4873
4874
4875
4876
4877
4878
4879
4880
4881
4882
4883
4884
4885
4886
4887
4888
4889
4890
4891
4892
4893
4894
4895
4896
4897
4898
4899
4900
4901
4902
4903
4904
4905
4906
4907
4908
4909
4910
4911
4912
4913
4914
4915
4916
4917
4918
4919
4920
4921
4922
4923
4924
4925
4926
4927
4928
4929
4930
4931
4932
4933
4934
4935
4936
4937
4938
4939
4940
4941
4942
4943
4944
4945
4946
4947
4948
4949
4950
4951
4952
4953
4954
4955
4956
4957
4958
4959
4960
4961
4962
4963
4964
4965
4966
4967
4968
4969
4970
4971
4972
4973
4974
4975
4976
4977
4978
4979
4980
4981
4982
4983
4984
4985
4986
4987
4988
4989
4990
4991
4992
4993
4994
4995
4996
4997
4998
4999
5000

```

```

procedure message;
(* this procedure prints the initial message to
the user*)
begin
  writeln;
  writeln('  : 8, 'WELCOME TO 78000 ASM ASSEMBLER');
  writeln;
  writeln('  : 8, 'To assemble a program :');
  writeln('  : 4, '1. Enter <filename> of source code (up to 15 chars)');
  writeln('  : 4, '2. Press <return> and wait....');
  writeln('  : 4, '3. Errors are stored in <filename>.o');
  writeln('  : 4, '4. Object code is stored in file 'executecode');
  writeln('  : 4, '5. If no errors occurred then type <ex> to start');
  writeln('  : 8, 'the simulator');
  writeln('  : 4, '6. To quit type U followed by a return ')
end; (* message *)

```



4323  
4324  
4325  
4326  
4327  
4328  
4329  
4330  
4331  
4332  
4333  
4334  
4335  
4336  
4337  
4338  
4339  
4340  
4341  
4342  
4343  
4344  
4345  
4346  
4347  
4348  
4349  
4350  
4351  
4352  
4353  
4354  
4355  
4356  
4357  
4358  
4359  
4360  
4361  
4362  
4363  
4364  
4365  
4366  
4367

```
procedure boot;  
  (* this procedure initializes all the variables and  
    the arrays. For the case of the arrays it uses files  
    where are written the initial data *)  
  
  var  
    count, index: integer;  
    init2 : text;  
  
  begin  
    controlreg(0) := 'FLAGS';  
    controlreg(1) := 'FCW';  
    controlreg(2) := 'RFFRES';  
    controlreg(3) := 'PSAPSE';  
    controlreg(4) := 'PSAPOF';  
    controlreg(5) := 'NSPSEG';  
    controlreg(6) := 'NSPOFF';  
    condition(1) := 'L';  
    condition(2) := 'LE';  
    condition(3) := 'ULF';  
    condition(4) := 'OV';  
    condition(5) := 'MI';  
    condition(6) := 'EU';  
    condition(7) := 'C';  
    condition(8) := 'TRUE';  
    condition(9) := 'GE';  
    condition(10) := 'GT';  
    condition(11) := 'UGI';  
    condition(12) := 'NOV';  
    condition(13) := 'PL';  
    condition(14) := 'NF';  
    condition(15) := 'UGE';  
    kindtypes(0) := 'ARRAY';  
    kindtypes(1) := 'BYTE';
```

```

4381 kindtypes[2] := 'WORD';
4382 kindtypes[3] := 'ההרה';
4383 kindtypes[4] := 'LONG';
4384 kindtypes[5] := 'LABEL';
4385 last := chr(0); (* is the last character in the input buffer *)
4386 cdp := 0;
4387 ll := 0;
4388 errorcounter := 0;
4389 ch := ' ';
4390 pc := 0;
4391 LAST := 0;
4392 (* clears the input buffer *)
4393 for index := 1 to 81 do
4394   empty[index] := ' ';
4395 card := empty;
4396 end2 := ' ', '!', chr(0), chr(9)];
4397 outfile := 'executecode';
4398 constindex := -1;
4399 varindex := -1;
4400 labelindex := 29;
4401 for index := 0 to maxlabel do
4402   (* initializes the variable and label table *)
4403   with variable[index] do begin
4404     name := margin;
4405     initial := -1;
4406     final := -1;
4407     size := -1;
4408   end;
4409 for index := stalabel to maxlabel do
4410   (* initializes the list of the labels for the
4411   forward reference *)
4412   for count := 0 to 20 do
4413     labellist[index, count] := -1;
4414   for index := 0 to 20 do
4415     with constlist[index] do begin

```

```

4444 name := merain;
4445 (* initializes the constant table *)
4446 value := -1;
4447 size := -1
4448 end;
4449
4450 reset(init, 'initial');
4451 reset(init1, 'reg');
4452 count := 1;
4453 while count <= maxon do begin
4454   (* initializes the opcode table reading for the
4455     input file init *)
4456   read(init, opcodefcount);
4457   count := count + 1
4458 end;
4459 count := 0;
4460 while count <= maxreq do begin
4461   (* initializes the register table reading
4462     from the input file init1 *)
4463   read(init1, registercount);
4464   count := count + 1
4465 end;
4466 for index := 0 to maxaddress do
4467   for count := 0 to 1 do
4468     memory[index][count] := '0';
4469   reset (init2, 'tit1?');
4470   (*opens the file with the link list of
4471     the opcodes *)
4472   for index := 1 to 185 do
4473     readln(init2, fastfindex);
4474   (* sets the array of the characters that
4475     gives the header of each basic character*)
4476   linkheader ['A'] := 1;
4477   linkheader ['B'] := 9;
4478   linkheader ['C'] := 12;
4479   linkheader ['0'] := 39;
4480
4481
4482
4483
4484
4485
4486
4487

```

```

4501
4502
4503
4504
4505
4506
4507
4508
4509
4510
4511
4512
4513
4514
4515
4516
4517
4518
4519
4520
4521
4522
4523
4524
4525
4526
4527
4528
4529
4530
4531
4532
4533
4534
4535
4536
4537
4538
4539
4540
4541

linkheader ['E'] := 48;
linkheader ['H'] := 55;
linkheader ['I'] := 57;
linkheader ['J'] := 71;
linkheader ['L'] := 74;
linkheader ['M'] := 96;
linkheader ['N'] := 99;
linkheader ['O'] := 103;
linkheader ['R'] := 116;
linkheader ['S'] := 136;
linkheader ['P'] := 131;
linkheader ['I'] := 167;
linkheader ['X'] := 183;
theend := false;
end; (* boot *)

```

```

4564 (* main program *)
4565
4566
4567 begin
4568     finished := false;
4569     boot;
4570     (* calls the procedure boot to initialize all the
4571        variables and tables *)
4572     message;
4573     (* prints the message to the user *)
4574     openfile;
4575     (* creates the input and output files *)
4576     load;
4577     constantanalyze;
4578     (* performs the analysis of the constants *)
4579     varanalyze;
4580     (* performs the analysis of the variables *)
4581     START := PC;
4582     (* it is the starting address of the program *)
4583     opcodeanalyze(finished);
4584     (* performs the analysis and encoding of the opcodes *)
4585     if not finished then
4586         errormessage(9)
4587     (* not normal termination of the program *)
4588     else if errorcounter = 0 then begin
4589         (* updates all the labels that have been referenced forward *)
4590         updatelabels;
4591         for on := 0 to PC do
4592             write(object, memoryloop[0], memoryloop[1]);
4593         (* transfers the generated code in the output file *)
4594         reset(object, 'executecode')
4595     end;
4596
4597     100:
4598     null;
4599     (*trace?*)
4600     if errorcounter = 0 then begin
4601
4602
4603
4604
4605
4606
4607

```

```

4621
4622
4623
4624
4625
4626
4627
4628
4629
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
4650
4651
4652
4653
4654

        writeln(' ': 4, 'To execute the program type < ex > and < cr >');
        write(' ': 4, 'Then type L and set the RPC to value = ');
        theend := true;
        dechex (trunc (START));
        end else
            writeln(' ': 4, 'Number of errors = ', errorcounter: 4,
                ' Open file ', errorfile )
        end. { assembler }

```

```

5
6
7
8 program initialize(input, output);
9
10 (* this program it opens the two files table1 and
11    table2 and it reads the data to create the opcode
12    and register tables for the assembler program. The
13    assembler it does not read from the files but in
14    the booting it loads the whole tables in the main
15    memory *)
16
17 const
18     blank = ' ';
19
20
21 type
22     bravo =
23     record
24         name: packed array [0..5] of char;
25         syntax: integer;
26         base : integer;
27     end;
28 (* this record is used to create the file of records of
29    opcodes *)
30     charlie =
31     record
32         name: packed array [0..5] of char;
33         value: integer;
34         length : integer;
35     end;
36 (* this record is used to create the file of records
37    of registers *)

```



```
61
62
63
64
65 var
66     count, index: integer;
67     pip, pop: text;
68     init: file of bravo;
69     init1: file of charlie;
70     opcode: bravo;
71     register: charlie;
72     ch: char;
```

```

125 begin
126     rewrite(init, 'initial');
127     reset(pip, 'table1');
128     (* it opens the table1 file and it also
129     creates the output file that is the default
130     file for the assembler program *)
131     opcode.name := blank;
132     count := 1;
133     while count <= 168 do begin
134         (* starts to read one by one opcode *)
135         index := 0;
136         read(pip, ch);
137         while (ch <> ' ') and (index <= 5) do begin
138             opcode.name[index] := ch;
139             (* it reads one by one character *)
140             index := index + 1;
141             read(pip, ch)
142         end;
143         readln(pip, opcode.syntax, opcode.base);
144         (* it reads the syntax and the base address of the opcode *)
145         write(init, opcode);
146         (* it transfers the opcode record to the output file *)
147         count := count + 1;
148         opcode.name := blank;
149         opcode.syntax := 0
150     end;
151     rewrite(init, 'reg');
152     (* it creates the register file for the assembler program *)
153     reset(pip, 'table2');
154     (* it opens the table2 *)
155     count := 1;
156     register.name := blank;
157     while count <= 44 do begin
158         index := 0;
159     end;
160
161
162
163
164
165
166
167
168

```

```

181
182
183
184
185 (* starts to read the name of the register one by one
186    character *)
187 repeat
188     read(pop, ch);
189     register.name(index) := ch;
190     index := index + 1;
191     until (ch = '.' ) or (index = 6);
192     readln (pop, register.value, register.length);
193     (* it reads the register value and the register length *)
194     write(initl, register);
195     (* it transfers the register record to the output file *)
196     register.name := blank;
197     register.value := 0;
198     register.length := 0;
199     count := count + 1;
200 end
201 end.

```

|    |       |   |   |       |
|----|-------|---|---|-------|
| 5  | HALT  | 0 | 0 | 31488 |
| 6  | IRFI  | 0 | 0 | 36103 |
| 7  | NOP   | 0 | 0 | 45056 |
| 8  | DAR   | 1 | 1 | 45322 |
| 9  | EXTS  | 1 | 1 | 45312 |
| 10 | FXTSR | 1 | 1 | 45319 |
| 11 | FXTSL | 1 | 1 | 40448 |
| 12 | RET   | 1 | 1 | 32512 |
| 13 | SC    | 1 | 1 | 3330  |
| 14 | NEG   | 1 | 1 | 3074  |
| 15 | NEGB  | 1 | 1 | 3328  |
| 16 | CUM   | 1 | 1 | 3072  |
| 17 | COMB  | 1 | 1 | 3332  |
| 18 | TEST  | 1 | 1 | 3076  |
| 19 | TESIR | 1 | 1 | 7168  |
| 20 | TESII | 1 | 1 | 3334  |
| 21 | TSFI  | 1 | 1 | 3078  |
| 22 | TSFIR | 1 | 1 | 7936  |
| 23 | CALL  | 1 | 1 | 53248 |
| 24 | CALR  | 1 | 1 | 256   |
| 25 | ADD   | 2 | 2 | 0     |
| 26 | ADDB  | 2 | 2 | 5032  |
| 27 | ADDL  | 2 | 2 | 2816  |
| 28 | CP    | 2 | 2 | 2560  |
| 29 | CPR   | 2 | 2 | 4096  |
| 30 | CPL   | 2 | 2 | 6912  |
| 31 | DIV   | 2 | 2 | 6056  |
| 32 | DIVL  | 2 | 2 | 6400  |
| 33 | MULT  | 2 | 2 | 6144  |
| 34 | MULTL | 2 | 2 | 768   |
| 35 | SUB   | 2 | 2 | 512   |
| 36 | SURB  | 2 | 2 | 4608  |
| 37 | SURL  | 2 | 2 | 1792  |
| 38 | AND   | 2 | 2 | 1536  |
| 39 | ANDB  | 2 | 2 |       |
| 40 |       |   |   |       |
| 41 |       |   |   |       |
| 42 |       |   |   |       |
| 43 |       |   |   |       |
| 44 |       |   |   |       |
| 45 |       |   |   |       |
| 46 |       |   |   |       |
| 47 |       |   |   |       |
| 48 |       |   |   |       |

|     |      |   |
|-----|------|---|
| 61  |      |   |
| 62  |      |   |
| 63  |      |   |
| 64  |      |   |
| 65  | NR   | 2 |
| 66  | NRB  | 2 |
| 67  | XOR  | 2 |
| 68  | XURH | 2 |
| 69  | LD   | 2 |
| 70  | LDR  | 2 |
| 71  | LDL  | 2 |
| 72  | FX   | 2 |
| 73  | FXB  | 2 |
| 74  | SBC  | 2 |
| 75  | SBCB | 2 |
| 76  | ADC  | 2 |
| 77  | ADCB | 2 |
| 78  | LDK  | 2 |
| 79  | LDAR | 2 |
| 80  | LDA  | 2 |
| 81  | RLDR | 2 |
| 82  | RRDB | 2 |
| 83  | RLB  | 2 |
| 84  | RL   | 2 |
| 85  | RLCB | 2 |
| 86  | RLC  | 2 |
| 87  | RRR  | 2 |
| 88  | RR   | 2 |
| 89  | RRCB | 2 |
| 90  | RRC  | 2 |
| 91  | SDAB | 2 |
| 92  | SDA  | 2 |
| 93  | SDAL | 2 |
| 94  | SDIB | 2 |
| 95  | SDL  | 2 |
| 96  | SDIL | 2 |
| 97  | DECB | 2 |
| 98  | DEC  | 2 |
| 99  | INCB | 2 |
| 100 |      |   |
| 101 |      |   |
| 102 |      |   |

1280  
1024  
2304  
2048  
8448  
8192  
5120  
11520  
11264  
46848  
46592  
46356  
46080  
48384  
13312  
30208  
48640  
48128  
45576  
45832  
45568  
45824  
45580  
45856  
45572  
45828  
45579  
45855  
45859  
45571  
45827  
45831  
10752  
11008  
10240

|     |        |   |       |
|-----|--------|---|-------|
| 125 | TNC    | 2 | 10496 |
| 126 | RTB    | 2 | 9128  |
| 127 | RTT    | 2 | 9984  |
| 128 | RESB   | 2 | 8704  |
| 129 | RES    | 2 | 8960  |
| 130 | SETB   | 2 | 9216  |
| 131 | SET    | 2 | 9472  |
| 132 | STORF  | 2 | 12032 |
| 133 | STORFB | 2 | 11776 |
| 134 | STORFL | 2 | 7424  |
| 135 | PUP    | 2 | 5888  |
| 136 | POPL   | 2 | 5376  |
| 137 | PUSH   | 2 | 4864  |
| 138 | PUSHL  | 2 | 4352  |
| 139 | JP     | 2 | 7680  |
| 140 | JK     | 2 | 57344 |
| 141 | TCCB   | 2 | 44544 |
| 142 | TCC    | 2 | 44800 |
| 143 | LDRB   | 2 | 12288 |
| 144 | LDR    | 2 | 12544 |
| 145 | LDRL   | 2 | 13568 |
| 146 | CLR    | 1 | 3336  |
| 147 | CLRB   | 1 | 3080  |
| 148 | LDM    | 3 | 7168  |
| 149 | DJNZ   | 2 | 61568 |
| 150 | DBJNZ  | 2 | 61440 |
| 151 | SLAB   | 2 | 45577 |
| 152 | SLA    | 2 | 45833 |
| 153 | SLAL   | 2 | 45837 |
| 154 | SLLB   | 2 | 45569 |
| 155 | SLI    | 2 | 45825 |
| 156 | SLLL   | 2 | 45829 |
| 157 | SRAB   | 2 | 45577 |
| 158 | SRA    | 2 | 45833 |
| 159 | SRAL   | 2 | 45837 |
| 160 |        |   |       |
| 161 |        |   |       |
| 162 |        |   |       |
| 163 |        |   |       |
| 164 |        |   |       |
| 165 |        |   |       |
| 166 |        |   |       |
| 167 |        |   |       |
| 168 |        |   |       |

|     |        |   |       |
|-----|--------|---|-------|
| 181 |        |   |       |
| 182 |        |   |       |
| 183 |        |   |       |
| 184 |        |   |       |
| 185 | SRLB   | 2 | 45509 |
| 186 | SRL    | 2 | 45825 |
| 187 | SRL    | 2 | 45829 |
| 188 | INR    | 2 | 15360 |
| 189 | TN     | 2 | 15610 |
| 190 | OUTB   | 2 | 15872 |
| 191 | OUT    | 2 | 16128 |
| 192 | LDCIB  | 2 | 35840 |
| 193 | LDCIL  | 2 | 32000 |
| 194 | CUMFLG | 2 | 36101 |
| 195 | RESFLG | 2 | 36097 |
| 196 | SETFLG | 2 | 36099 |
| 197 | LPDS   | 1 | 14592 |
| 198 | INDB   | 3 | 14856 |
| 199 | IND    | 3 | 15112 |
| 200 | INDRR  | 3 | 14856 |
| 201 | INDR   | 3 | 15112 |
| 202 | INIB   | 3 | 14848 |
| 203 | INI    | 3 | 15104 |
| 204 | INIRB  | 3 | 14848 |
| 205 | INIR   | 3 | 15104 |
| 206 | OUTDR  | 3 | 14858 |
| 207 | OUTD   | 3 | 15114 |
| 208 | OUTIB  | 3 | 14850 |
| 209 | OUTI   | 3 | 15106 |
| 210 | OUTRR  | 3 | 14858 |
| 211 | OUTR   | 3 | 15114 |
| 212 | OUTRB  | 3 | 14850 |
| 213 | OUTR   | 3 | 15106 |
| 214 | LDDR   | 3 | 47025 |
| 215 | LDD    | 3 | 47881 |
| 216 | LDDRR  | 3 | 47025 |
| 217 | LDDR   | 3 | 47881 |
| 218 | LUIB   | 3 | 47617 |
| 219 | LUI    | 3 | 47873 |
| 220 |        |   |       |
| 221 |        |   |       |
| 222 |        |   |       |



|     |        |   |       |
|-----|--------|---|-------|
| 245 | LDIRB  | 3 | 47617 |
| 246 | LDIR   | 3 | 47873 |
| 247 | TRDB   | 3 | 47112 |
| 248 | TRDKB  | 3 | 47116 |
| 249 | TRIB   | 3 | 47104 |
| 250 | TRIKB  | 3 | 47108 |
| 251 | TRIDB  | 3 | 47114 |
| 252 | TRIDRB | 3 | 47118 |
| 253 | TRTIR  | 3 | 47100 |
| 254 | TRTIRB | 3 | 47110 |
| 255 | CPDB   | 4 | 47624 |
| 256 | CPD    | 4 | 47880 |
| 257 | CPDKB  | 4 | 47628 |
| 258 | CPDR   | 4 | 47884 |
| 259 | CPIB   | 4 | 47616 |
| 260 | CPI    | 4 | 47872 |
| 261 | CPIRB  | 4 | 47620 |
| 262 | CPIR   | 4 | 47876 |
| 263 | CPSDR  | 4 | 47626 |
| 264 | CPSD   | 4 | 47882 |
| 265 | CPSDRB | 4 | 47630 |
| 266 | CPSDR  | 4 | 47886 |
| 267 | CPSIR  | 4 | 47618 |
| 268 | CPSI   | 4 | 47874 |
| 269 | CPSIRB | 4 | 47622 |
| 270 | CPSIR  | 4 | 47878 |
| 271 | DI     | 1 | 51744 |
| 272 | EI     | 1 | 51748 |

|     |  |  |  |
|-----|--|--|--|
| 273 |  |  |  |
| 274 |  |  |  |
| 275 |  |  |  |
| 276 |  |  |  |
| 277 |  |  |  |
| 278 |  |  |  |
| 279 |  |  |  |
| 280 |  |  |  |
| 281 |  |  |  |
| 282 |  |  |  |
| 283 |  |  |  |
| 284 |  |  |  |
| 285 |  |  |  |
| 286 |  |  |  |
| 287 |  |  |  |
| 288 |  |  |  |

[illegible]

|    |      |    |   |
|----|------|----|---|
| 64 |      |    |   |
| 65 | RL3  | 11 | 0 |
| 66 | RL4  | 12 | 0 |
| 67 | RL5  | 13 | 0 |
| 68 | RL6  | 14 | 0 |
| 69 | RL7  | 15 | 0 |
| 70 | RU0  | 0  | 4 |
| 71 | RU4  | 4  | 4 |
| 72 | RU8  | 8  | 4 |
| 73 | RU12 | 12 | 4 |
| 74 |      |    |   |
| 75 |      |    |   |
| 76 |      |    |   |
| 77 |      |    |   |
| 78 |      |    |   |
| 79 |      |    |   |
| 80 |      |    |   |
| 81 |      |    |   |
| 82 |      |    |   |
| 83 |      |    |   |
| 84 |      |    |   |
| 85 |      |    |   |
| 86 |      |    |   |
| 87 |      |    |   |
| 88 |      |    |   |
| 89 |      |    |   |
| 90 |      |    |   |
| 91 |      |    |   |
| 92 |      |    |   |
| 93 |      |    |   |
| 94 |      |    |   |

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42

This is the table that provides the link lists  
of the opcodes by alphabetical order and so it reduces  
the search. For example if the opcode starts from  
the character H the program compares the opcode only  
with the opcode HALT.

A

21  
22  
23  
34  
35  
47  
48  
-1  
72  
73  
-1  
12  
13  
19  
20  
24  
25  
26  
115  
151

B

C

64  
 65  
 66  
 67  
 68  
 69  
 70  
 71  
 72  
 73  
 74  
 75  
 76  
 77  
 78  
 79  
 80  
 81  
 82  
 83  
 84  
 85  
 86  
 87  
 88  
 89  
 90  
 91  
 92  
 93  
 94  
 95  
 96  
 97  
 98  
 99  
 100  
 101  
 102  
 103  
 104  
 105  
 106  
 107  
 108  
 109  
 110  
 111  
 112  
 113  
 114  
 115  
 116  
 117  
 118  
 119  
 120  
 121  
 122  
 123  
 124  
 125  
 126  
 127  
 128  
 129  
 130  
 131  
 132  
 133  
 134  
 135  
 136  
 137  
 138  
 139  
 140  
 141  
 142  
 143  
 144  
 145  
 146  
 147  
 148  
 149  
 150  
 151  
 152  
 153  
 154  
 155  
 156  
 157  
 158  
 159  
 160  
 161  
 162  
 163  
 164  
 165  
 166  
 167  
 168  
 169  
 170  
 171  
 172  
 173  
 174  
 175  
 176  
 177  
 178  
 179  
 180  
 181  
 182  
 183  
 184  
 185  
 186  
 187  
 188  
 189  
 190  
 191  
 192  
 193  
 194  
 195  
 196  
 197  
 198  
 199  
 200  
 201  
 202  
 203  
 204  
 205  
 206  
 207  
 208  
 209  
 210  
 211  
 212  
 213  
 214  
 215  
 216  
 217  
 218  
 219  
 220  
 221  
 222  
 223  
 224  
 225  
 226  
 227  
 228  
 229  
 230  
 231  
 232  
 233  
 234  
 235  
 236  
 237  
 238  
 239  
 240  
 241  
 242  
 243  
 244  
 245  
 246  
 247  
 248  
 249  
 250  
 251  
 252  
 253  
 254  
 255  
 256  
 257  
 258  
 259  
 260  
 261  
 262  
 263  
 264  
 265  
 266  
 267  
 268  
 269  
 270  
 271  
 272  
 273  
 274  
 275  
 276  
 277  
 278  
 279  
 280  
 281  
 282  
 283  
 284  
 285  
 286  
 287  
 288  
 289  
 290  
 291  
 292  
 293  
 294  
 295  
 296  
 297  
 298  
 299  
 300  
 301  
 302  
 303  
 304  
 305  
 306  
 307  
 308  
 309  
 310  
 311  
 312  
 313  
 314  
 315  
 316  
 317  
 318  
 319  
 320  
 321  
 322  
 323  
 324  
 325  
 326  
 327  
 328  
 329  
 330  
 331  
 332  
 333  
 334  
 335  
 336  
 337  
 338  
 339  
 340  
 341  
 342  
 343  
 344  
 345  
 346  
 347  
 348  
 349  
 350  
 351  
 352  
 353  
 354  
 355  
 356  
 357  
 358  
 359  
 360  
 361  
 362  
 363  
 364  
 365  
 366  
 367  
 368  
 369  
 370  
 371  
 372  
 373  
 374  
 375  
 376  
 377  
 378  
 379  
 380  
 381  
 382  
 383  
 384  
 385  
 386  
 387  
 388  
 389  
 390  
 391  
 392  
 393  
 394  
 395  
 396  
 397  
 398  
 399  
 400  
 401  
 402  
 403  
 404  
 405  
 406  
 407  
 408  
 409  
 410  
 411  
 412  
 413  
 414  
 415  
 416  
 417  
 418  
 419  
 420  
 421  
 422  
 423  
 424  
 425  
 426  
 427  
 428  
 429  
 430  
 431  
 432  
 433  
 434  
 435  
 436  
 437  
 438  
 439  
 440  
 441  
 442  
 443  
 444  
 445  
 446  
 447  
 448  
 449  
 450  
 451  
 452  
 453  
 454  
 455  
 456  
 457  
 458  
 459  
 460  
 461  
 462  
 463  
 464  
 465  
 466  
 467  
 468  
 469  
 470  
 471  
 472  
 473  
 474  
 475  
 476  
 477  
 478  
 479  
 480  
 481  
 482  
 483  
 484  
 485  
 486  
 487  
 488  
 489  
 490  
 491  
 492  
 493  
 494  
 495  
 496  
 497  
 498  
 499  
 500  
 501  
 502  
 503  
 504  
 505  
 506  
 507  
 508  
 509  
 510  
 511  
 512  
 513  
 514  
 515  
 516  
 517  
 518  
 519  
 520  
 521  
 522  
 523  
 524  
 525  
 526  
 527  
 528  
 529  
 530  
 531  
 532  
 533  
 534  
 535  
 536  
 537  
 538  
 539  
 540  
 541  
 542  
 543  
 544  
 545  
 546  
 547  
 548  
 549  
 550  
 551  
 552  
 553  
 554  
 555  
 556  
 557  
 558  
 559  
 560  
 561  
 562  
 563  
 564  
 565  
 566  
 567  
 568  
 569  
 570  
 571  
 572  
 573  
 574  
 575  
 576  
 577  
 578  
 579  
 580  
 581  
 582  
 583  
 584  
 585  
 586  
 587  
 588  
 589  
 590  
 591  
 592  
 593  
 594  
 595  
 596  
 597  
 598  
 599  
 600  
 601  
 602  
 603  
 604  
 605  
 606  
 607  
 608  
 609  
 610  
 611  
 612  
 613  
 614  
 615  
 616  
 617  
 618  
 619  
 620  
 621  
 622  
 623  
 624  
 625  
 626  
 627  
 628  
 629  
 630  
 631  
 632  
 633  
 634  
 635  
 636  
 637  
 638  
 639  
 640  
 641  
 642  
 643  
 644  
 645  
 646  
 647  
 648  
 649  
 650  
 651  
 652  
 653  
 654  
 655  
 656  
 657  
 658  
 659  
 660  
 661  
 662  
 663  
 664  
 665  
 666  
 667  
 668  
 669  
 670  
 671  
 672  
 673  
 674  
 675  
 676  
 677  
 678  
 679  
 680  
 681  
 682  
 683  
 684  
 685  
 686  
 687  
 688  
 689  
 690  
 691  
 692  
 693  
 694  
 695  
 696  
 697  
 698  
 699  
 700  
 701  
 702  
 703  
 704  
 705  
 706  
 707  
 708  
 709  
 710  
 711  
 712  
 713  
 714  
 715  
 716  
 717  
 718  
 719  
 720  
 721  
 722  
 723  
 724  
 725  
 726  
 727  
 728  
 729  
 730  
 731  
 732  
 733  
 734  
 735  
 736  
 737  
 738  
 739  
 740  
 741  
 742  
 743  
 744  
 745  
 746  
 747  
 748  
 749  
 750  
 751  
 752  
 753  
 754  
 755  
 756  
 757  
 758  
 759  
 760  
 761  
 762  
 763  
 764  
 765  
 766  
 767  
 768  
 769  
 770  
 771  
 772  
 773  
 774  
 775  
 776  
 777  
 778  
 779  
 780  
 781  
 782  
 783  
 784  
 785  
 786  
 787  
 788  
 789  
 790  
 791  
 792  
 793  
 794  
 795  
 796  
 797  
 798  
 799  
 800  
 801  
 802  
 803  
 804  
 805  
 806  
 807  
 808  
 809  
 810  
 811  
 812  
 813  
 814  
 815  
 816  
 817  
 818  
 819  
 820  
 821  
 822  
 823  
 824  
 825  
 826  
 827  
 828  
 829  
 830  
 831  
 832  
 833  
 834  
 835  
 836  
 837  
 838  
 839  
 840  
 841  
 842  
 843  
 844  
 845  
 846  
 847  
 848  
 849  
 850  
 851  
 852  
 853  
 854  
 855  
 856  
 857  
 858  
 859  
 860  
 861  
 862  
 863  
 864  
 865  
 866  
 867  
 868  
 869  
 870  
 871  
 872  
 873  
 874  
 875  
 876  
 877  
 878  
 879  
 880  
 881  
 882  
 883  
 884  
 885  
 886  
 887  
 888  
 889  
 890  
 891  
 892  
 893  
 894  
 895  
 896  
 897  
 898  
 899  
 900  
 901  
 902  
 903  
 904  
 905  
 906  
 907  
 908  
 909  
 910  
 911  
 912  
 913  
 914  
 915  
 916  
 917  
 918  
 919  
 920  
 921  
 922  
 923  
 924  
 925  
 926  
 927  
 928  
 929  
 930  
 931  
 932  
 933  
 934  
 935  
 936  
 937  
 938  
 939  
 940  
 941  
 942  
 943  
 944  
 945  
 946  
 947  
 948  
 949  
 950  
 951  
 952  
 953  
 954  
 955  
 956  
 957  
 958  
 959  
 960  
 961  
 962  
 963  
 964  
 965  
 966  
 967  
 968  
 969  
 970  
 971  
 972  
 973  
 974  
 975  
 976  
 977  
 978  
 979  
 980  
 981  
 982  
 983  
 984  
 985  
 986  
 987  
 988  
 989  
 990  
 991  
 992  
 993  
 994  
 995  
 996  
 997  
 998  
 999  
 1000  
 1001  
 1002  
 1003  
 1004  
 1005  
 1006  
 1007  
 1008  
 1009  
 1010  
 1011  
 1012  
 1013  
 1014  
 1015  
 1016  
 1017  
 1018  
 1019  
 1020  
 1021  
 1022  
 1023  
 1024  
 1025  
 1026  
 1027  
 1028  
 1029  
 1030  
 1031  
 1032  
 1033  
 1034  
 1035  
 1036  
 1037  
 1038  
 1039  
 1040  
 1041  
 1042  
 1043  
 1044  
 1045  
 1046  
 1047  
 1048  
 1049  
 1050  
 1051  
 1052  
 1053  
 1054  
 1055  
 1056  
 1057  
 1058  
 1059  
 1060  
 1061  
 1062  
 1063  
 1064  
 1065  
 1066  
 1067  
 1068  
 1069  
 1070  
 1071  
 1072  
 1073  
 1074  
 1075  
 1076  
 1077  
 1078  
 1079  
 1080  
 1081  
 1082  
 1083  
 1084  
 1085  
 1086  
 1087  
 1088  
 1089  
 1090  
 1091  
 1092  
 1093  
 1094  
 1095  
 1096  
 1097  
 1098  
 1099  
 1100  
 1101  
 1102  
 1103  
 1104  
 1105  
 1106  
 1107  
 1108  
 1109  
 1110  
 1111  
 1112  
 1113  
 1114  
 1115  
 1116  
 1117  
 1118  
 1119  
 1120  
 1121  
 1122  
 1123  
 1124  
 1125  
 1126  
 1127  
 1128  
 1129  
 1130  
 1131  
 1132  
 1133  
 1134  
 1135  
 1136  
 1137  
 1138  
 1139  
 1140  
 1141  
 1142  
 1143  
 1144  
 1145  
 1146  
 1147  
 1148  
 1149  
 1150  
 1151  
 1152  
 1153  
 1154  
 1155  
 1156  
 1157  
 1158  
 1159  
 1160  
 1161  
 1162  
 1163  
 1164  
 1165  
 1166  
 1167  
 1168  
 1169  
 1170  
 1171  
 1172  
 1173  
 1174  
 1175  
 1176  
 1177  
 1178  
 1179  
 1180  
 1181  
 1182  
 1183  
 1184  
 1185  
 1186  
 1187  
 1188  
 1189  
 1190  
 1191  
 1192  
 1193  
 1194  
 1195  
 1196  
 1197  
 1198  
 1199  
 1200  
 1201  
 1202  
 1203  
 1204  
 1205  
 1206  
 1207  
 1208  
 1209  
 1210  
 1211  
 1212  
 1213  
 1214  
 1215  
 1216  
 1217  
 1218  
 1219  
 1220  
 1221  
 1222  
 1223  
 1224  
 1225  
 1226  
 1227  
 1228  
 1229  
 1230  
 1231  
 1232  
 1233  
 1234  
 1235  
 1236  
 1237  
 1238  
 1239  
 1240  
 1241  
 1242  
 1243  
 1244  
 1245  
 1246  
 1247  
 1248  
 1249  
 1250  
 1251  
 1252  
 1253  
 1254  
 1255  
 1256  
 1257  
 1258  
 1259  
 1260  
 1261  
 1262  
 1263  
 1264  
 1265  
 1266  
 1267  
 1268  
 1269  
 1270  
 1271  
 1272  
 1273  
 1274  
 1275  
 1276  
 1277  
 1278  
 1279  
 1280  
 1281  
 1282  
 1283  
 1284  
 1285  
 1286  
 1287  
 1288  
 1289  
 1290  
 1291  
 1292  
 1293  
 1294  
 1295  
 1296  
 1297  
 1298  
 1299  
 1300  
 1301  
 1302  
 1303  
 1304  
 1305  
 1306  
 1307  
 1308  
 1309  
 1310  
 1311  
 1312  
 1313  
 1314  
 1315  
 1316  
 1317  
 1318  
 1319  
 1320  
 1321  
 1322  
 1323  
 1324  
 1325  
 1326  
 1327  
 1328  
 1329  
 1330  
 1331  
 1332  
 1333  
 1334  
 1335  
 1336  
 1337  
 1338  
 1339  
 1340  
 1341  
 1342  
 1343  
 1344  
 1345  
 1346  
 1347  
 1348  
 1349  
 1350  
 1351  
 1352  
 1353  
 1354  
 1355  
 1356  
 1357  
 1358  
 1359  
 1360  
 1361  
 1362  
 1363  
 1364  
 1365  
 1366  
 1367  
 1368  
 1369  
 1370  
 1371  
 1372  
 1373  
 1374  
 1375  
 1376  
 1377  
 1378  
 1379  
 1380  
 1381  
 1382  
 1383  
 1384  
 1385  
 1386  
 1387  
 1388  
 1389  
 1390  
 1391  
 1392  
 1393  
 1394  
 1395  
 1396  
 1397  
 1398  
 1399  
 1400  
 1401  
 1402  
 1403  
 1404  
 1405  
 1406  
 1407  
 1408  
 1409  
 1410  
 1411  
 1412  
 1413  
 1414  
 1415  
 1416  
 1417  
 1418  
 1419  
 1420  
 1421  
 1422  
 1423  
 1424  
 1425  
 1426  
 1427  
 1428  
 1429  
 1430  
 1431  
 1432  
 1433  
 1434  
 1435  
 1436  
 1437  
 1438  
 1439  
 1440  
 1441  
 1442  
 1443  
 1444  
 1445  
 1446  
 1447  
 1448  
 1449  
 1450  
 1451  
 1452  
 1453  
 1454  
 1455  
 1456  
 1457  
 1458  
 1459  
 1460  
 1461  
 1462  
 1463  
 1464  
 1465  
 1466  
 1467  
 1468  
 1469  
 1470  
 1471  
 1472  
 1473  
 1474  
 1475  
 1476  
 1477  
 1478  
 1479  
 1480  
 1481  
 1482  
 1483  
 1484  
 1485  
 1486  
 1487  
 1488  
 1489  
 1490  
 1491  
 1492  
 1493  
 1494  
 1495  
 1496  
 1497  
 1498  
 1499  
 1500  
 1501  
 1502  
 1503  
 1504  
 1505  
 1506  
 1507  
 1508  
 1509  
 1510  
 1511  
 1512  
 1513  
 1514  
 1515  
 1516  
 1517  
 1518  
 1519  
 1520  
 1521  
 1522  
 1523  
 1524  
 1525  
 1526  
 1527  
 1528  
 1529  
 1530  
 1531  
 1532  
 1533  
 1534  
 1535  
 1536  
 1537  
 1538  
 1539  
 1540  
 1541  
 1542  
 1543  
 1544

121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163

-1  
70  
71  
109  
110  
119  
120  
2  
121  
122  
123  
124  
125  
126  
-1  
85  
86  
-1  
40  
41  
42  
49  
50  
51  
94  
89  
90  
91  
113  
114  
118  
135  
136  
137  
138

T

J

L

184  
 185  
 186  
 187  
 188  
 189  
 190  
 191  
 192  
 193  
 194  
 195  
 196  
 197  
 198  
 199  
 200  
 201  
 202  
 203  
 204  
 205  
 206  
 207  
 208  
 209  
 210  
 211  
 212  
 213  
 214  
 215  
 216  
 217  
 218  
 219  
 220  
 221  
 222  
 223  
 224  
 225  
 226  
 227  
 228  
 139  
 140  
 141  
 142  
 -1  
 29  
 30  
 -1  
 10  
 11  
 3  
 -1  
 36  
 37  
 111  
 112  
 127  
 128  
 129  
 130  
 131  
 132  
 133  
 134  
 -1  
 R  
 52  
 53  
 54  
 55  
 56  
 57  
 58  
 59  
 60

M

N

O

R



|     |     |
|-----|-----|
| 241 | 61  |
| 242 | 74  |
| 243 | 75  |
| 244 | 116 |
| 245 | -1  |
| 246 | 81  |
| 247 | 82  |
| 248 | 83  |
| 249 | 84  |
| 250 | -1  |
| 251 | 9   |
| 252 | 31  |
| 253 | 32  |
| 254 | 33  |
| 255 | 45  |
| 256 | 46  |
| 257 | 62  |
| 258 | 63  |
| 259 | 64  |
| 260 | 65  |
| 261 | 66  |
| 262 | 67  |
| 263 | 70  |
| 264 | 71  |
| 265 | 78  |
| 266 | 79  |
| 267 | 80  |
| 268 | 97  |
| 269 | 98  |
| 270 | 99  |
| 271 | 100 |
| 272 | 101 |
| 273 | 102 |
| 274 | 103 |
| 275 | 104 |
| 276 |     |
| 277 |     |
| 278 |     |
| 279 |     |
| 280 |     |
| 281 |     |
| 282 |     |

P

S

305  
 306  
 307  
 308  
 309  
 310  
 311  
 312  
 313  
 314  
 315  
 316  
 317  
 318  
 319  
 320  
 321  
 322  
 323  
 324  
 325  
 326  
 327  
 328  
 329  
 330  
 331  
 332  
 333  
 334  
 335  
 336  
 337  
 338  
 339  
 340  
 341  
 342  
 343  
 344  
 345  
 346  
 347  
 348  
 349

T

X

```

1
2
3
4
5
6
7
8 : RUBLE SORT MODULE
9
10
11 :CONSTANT DECLARATIONS
12 FALSE := 0
13 TRUE1 := 1
14 VARIA :VARIABLE DECLARATIONS
15 ORG %0100
16 LIST ARRAY (10 WORD) := 0,1,2,4,3,5,6,7,9,8
17 SWITCH BYIF := 0
18 T1 LABEL
19 T2 LABEL
20 T3 LABEL
21 SORT LABEL
22 T5 LABEL
23 ENTRY
24 LD R0, #18 :INITIALIZE LOOP CONTROL
25 CALL SORT :CALL SORT MODULE
26 HALT : END OF PROGRAM
27 ORG %0200
28
29
30 SORT : STURER SWITCH , #FALSE ! INITIALIZE
31 : THE SWITCH
32 CLR R1 : CLEAR ARRAY POINTER
33 CP R1,R0 :DONE ?
34 JP C , T1 : IF CARRY THEN EXIT
35 JP TRUEF , T2
36 T1 : LD R2,R1 : INITIALIZE POINTER
37 INC R2, #2 : J := J + 1
38 LD R4, LIST (R1) : LOAD THE MEMORY CONTENTS
39 : IN REGISTERS
40
41
42
43

```

```

65 LD R6, LIST (R2)
66 CP R4,R6 :IF LIST11 > LIST1J THEN
67 JP ULE , I3 :EXCHANGE
68 STOREB SWITCH , #TRUE1 :STORE THAT CHANGE
69 : HAS OCCURED
70 STORE LIST (P1), R6 : INTERCHANGE CONTENTS
71 : OF MEMORY
72 STORE LIST(R2), R4
73 T3 : INC R1, #2 :ADVANCE POINIER
74 JR TRUE , $+ZFB :END OF LOOP
75 T2 : CPB SWITCH , #FALSE :CHECK CONDITION
76 :OF SWITCH
77 JP NE ,T5
78 RET TRUE
79 T5 : JR TRUE, $+ZE0
80 END
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101

```

SOURCE FILE

-----

.PC CUDE

! RUBBLE SORT MODULE

CONSTA !CONSTANT DECLARATIONS

FALSE := 0

TRUE1 := 1

VARIA !VARIABLE DECLARATIONS

ORC %0100

LIST ARRAY (10 WORD) := 0,1,2,4,3,5,6,7,9,8

SWITCH BYTE := 0

T1 LABEL

T2 LABEL

T3 LABEL

SORT LABEL

|     |  |  |  |  |  |
|-----|--|--|--|--|--|
| 64  |  |  |  |  |  |
| 65  |  |  |  |  |  |
| 66  |  |  |  |  |  |
| 67  |  |  |  |  |  |
| 68  |  |  |  |  |  |
| 69  |  |  |  |  |  |
| 70  |  |  |  |  |  |
| 71  |  |  |  |  |  |
| 72  |  |  |  |  |  |
| 73  |  |  |  |  |  |
| 74  |  |  |  |  |  |
| 75  |  |  |  |  |  |
| 76  |  |  |  |  |  |
| 77  |  |  |  |  |  |
| 78  |  |  |  |  |  |
| 79  |  |  |  |  |  |
| 80  |  |  |  |  |  |
| 81  |  |  |  |  |  |
| 82  |  |  |  |  |  |
| 83  |  |  |  |  |  |
| 84  |  |  |  |  |  |
| 85  |  |  |  |  |  |
| 86  |  |  |  |  |  |
| 87  |  |  |  |  |  |
| 88  |  |  |  |  |  |
| 89  |  |  |  |  |  |
| 90  |  |  |  |  |  |
| 91  |  |  |  |  |  |
| 92  |  |  |  |  |  |
| 93  |  |  |  |  |  |
| 94  |  |  |  |  |  |
| 95  |  |  |  |  |  |
| 96  |  |  |  |  |  |
| 97  |  |  |  |  |  |
| 98  |  |  |  |  |  |
| 99  |  |  |  |  |  |
| 100 |  |  |  |  |  |
| 101 |  |  |  |  |  |
| 102 |  |  |  |  |  |
| 103 |  |  |  |  |  |
| 104 |  |  |  |  |  |
| 105 |  |  |  |  |  |
| 106 |  |  |  |  |  |
| 107 |  |  |  |  |  |
| 108 |  |  |  |  |  |

```

IS LABEL
ENTRY
LD R0, #18 ;INITIALIZE LOOP CONTROL

CALL SORT ;CALL SORT MODULE

HALT ; END OF PROGRAM

ORG %0200

SORT : STOREB SWITCH , #FALSE ; INITIALIZE

      ; THE SWITCH
CLR R1 ; CLEAR ARRAY POINTER
CP R1,R0 ;DONE ?
JP C , T1 ; IF CARRY THEN EXIT

JP TRUE , T2

T1 : LD R2,R1 ; INITIALIZE POINTER
INC R2, #2 ; J := J + 1
LD R4, LIST (R1) ; LOAD THE MEMORY CONTENTS

```

```

121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162

021A 0126
021C 0100
021E 8B64
0220 5F03
0222 0000
0224 4C05
0226 0114
0228 0100
022A 6F16
022C 0100
022E 6F24
0230 0100
0232 A911
0234 EAE9
0236 4C01
0238 0114
023A 0000
023C 5F0E
023E 0000
0240 9F08

LD R6, LIST (R2)
; IN REGISTERS

CP R4,R6 ;IF LIST(1) > LIST(J) THEN
JP UL , T3 ;EXCHANGE

STOREB SWITCH , #TRUE1 ;STORE THAT CHANGE

; HAS OCCURED
STORE LIST (R1), R6 ; INTERCHANGE CONIENTS

STORE LIST(R2), R4
; OF MEMORY

T3 : INC R1, #2 ;ADVANCE POINIER
JR TRUE , $+2EB ;END OF LOOP

T2 : CPB SWITCH ,#FALSE ;CHECK CONDITION

JP NF ,15
;OF SWITCH

RET TRUE

```



184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215

0242 E8DF

IS : JP TRUE, \$+ZE0

FND

UPDATE THE FORWARD REFERENCED LABELS

020C 0212  
0210 0236  
0222 0232  
011C 0200  
023E 0242

```

1
2
3
4
5 CONSTA
6 NEW := 12
7 BETA := %12
8 VARIA
9 ONE BYTE := %33
10 TWO WORD := %AF23
11 T2 LONG := %AE12
12 T3 ARRAY ( 12 BYTE ) := 00,01,02,03,04,05,06,07.
13 T4 LABEL
14 T5 LABEL
15 ENTRY
16 CALK $+0 ! 0004
17 ADD R1, R2 ! 8121
18 ADDB RH1, #0, ! 0001 3000
19 ADDL RR2, %0010 ( R1 ) ! 5612 0010
20 CP R1, R3 ! 8B21
21 CPH RH2, #23 ! 0A02 1700
22 CPL RR4, T3 (4) ! 5004 000C
23 CP @RB, #0020 ! 0D81 0014
24 DIV RK2, R4 ! 9B42
25 DIVL R04, TWO ! 5A04 0002
26 MULI KR2, R6 ! 9962
27 MULIL RQR, #%0011 ! 1808 0000 0011
28 SUBB R10, PH0 ! ! 8208
29 SUR R1, #%0A ! 0301 000A
30 SUBL KR2, RK4 ! 9242
31 ANDB R10, PL1 ! 8698
32 AND R2, TWO ! 4702 0002
33 ORR RH1, T3 (4) ! 4401 0C00
34 OR R1, T3 (R2) ! 4521 0008
35 XORB RH1, RH2 ! 8821
36 XOR P1, @RK2 ! 0921
37 LDB RH1, #%1234 ! 2001 3400
38 LD R2, #%1234 ! 2101 1234
39 LDI RK2, %0400 ! 5402 0400
40
41
42

```

```

65 FX R1,R3 ; AD31
66 FXR PL1 ,WR3 ; 2C93
67 SBC R2,R4 ; B742
68 SBCB RH0,RH1 ; B610
69 ADC R7,R6 ; B567
70 ADCB RH2,RH4 ; B442
71 LOK R0, #ZA ; B00A
72 LDAR R1, I4 ; 3401 007A
73 LDA R8 ,%0080 ; 7608 0080
74 RLDB RH1,RH2 ; 6E21
75 T4 : RRDB PH1,RH3 ; RC31
76 RLB RL1, #2 ; B29A
77 RL R1 ;B318
78 RLDB RH2, #2 ; B222
79 RLC R8 ; B380
80 RRR RH2 ; B22C
81 RR R7 ,#2 ;B37F
82 RRCB RH2 ;B224
83 RRC R3 ;B334
84 SDA R2,R1 ; B32B 0001
85 SDAH RH2,RH1 ; B22B 0001
86 SDAL RR2,RR4 ; B32F 0004
87 SDL R2,R1 ;B323 0001
88 SDB RL2,R1 ; B243 0001
89 SDLL RR4 , R1 ;B347 0001
90 ADDL RR2, #%01011234 ; 1602 01011234
91 SUBL RR4 , #%00010001 ; 1204 00010001
92 D1VL R00 , #%11111111 ; 1A00 11111111
93 MUL11 RQ4, #%02020202 ; 1804 02020202
94 LDI RRR, #%0001000A ; 1408 0001000A
95 JR TRUE, B+%12 ; L810
96 FND
97
98
99
100
101
102
103
104
105
106
107
108

```

SOURCE FILE

-----

PC CODE

CONSTA

NEW := 12

RETA := %12

VARIA

ONE BYTE := %53

TWO WORD := %AF23

T2 LONG := %AE12

T3 ARRAY [ 12 BYTE ] := 00,01,02,03,04,05,06,07.

T4 LABEL

T5 LABEL

FNTRY

CALR \$+6 ; 0004

0020 0004

|     |               |                                     |  |
|-----|---------------|-------------------------------------|--|
| 64  |               |                                     |  |
| 65  | 0022 8121     | ADD R1, R2 ; 8121                   |  |
| 66  |               |                                     |  |
| 67  | 0024 0001     | ADDB RH1 , #'0' ; 0001 3000         |  |
| 68  | 0026 3000     |                                     |  |
| 69  |               |                                     |  |
| 70  | 0028 5612     | ADDL RR2, %0010 ( R1 ) ; 5612 0010  |  |
| 71  | 002A 0010     |                                     |  |
| 72  |               |                                     |  |
| 73  | 002C 8B31     | CP R1,R3 ; 8B21                     |  |
| 74  |               |                                     |  |
| 75  | 002E 0A02     | CPB RH2,#23 ; 0A02 1700             |  |
| 76  | 0030 1700     |                                     |  |
| 77  |               |                                     |  |
| 78  | 0032 5004     | CPI RR4 , T3 (4) ; 5004 000C        |  |
| 79  | 0034 000C     |                                     |  |
| 80  |               |                                     |  |
| 81  | 0036 0D81     | CP wPB , #0020 ; 0D81 0014          |  |
| 82  | 0038 0014     |                                     |  |
| 83  |               |                                     |  |
| 84  | 003A 9B42     | DIV RR2, R4 ; 9B42                  |  |
| 85  |               |                                     |  |
| 86  | 003C 5A04     | DIVL RQ4 , IWO ; 5A04 0002          |  |
| 87  | 003E 0002     |                                     |  |
| 88  |               |                                     |  |
| 89  | 0040 9962     | MULF RP2 , R6 ; 9962                |  |
| 90  |               |                                     |  |
| 91  | 0042 1908     | MUL IL RQH, #%0011 ; 1808 0000 0011 |  |
| 92  | 0044 00000011 |                                     |  |
| 93  |               |                                     |  |
| 94  | 0048 8208     | SURB RL0 ,RH0 ; ; 8208              |  |
| 95  |               |                                     |  |
| 96  | 004A 0301     | SUR R1 , #%0A ; 0301 000A           |  |
| 97  | 004C 000A     |                                     |  |
| 98  |               |                                     |  |
| 99  |               | SURL RR2, RR4 ; 9242                |  |
| 100 |               |                                     |  |
| 101 |               |                                     |  |
| 102 |               |                                     |  |
| 103 |               |                                     |  |
| 104 |               |                                     |  |
| 105 |               |                                     |  |
| 106 |               |                                     |  |
| 107 |               |                                     |  |
| 108 |               |                                     |  |

|     |           |
|-----|-----------|
| 121 |           |
| 122 |           |
| 123 |           |
| 124 |           |
| 125 | 004E 9242 |
| 126 |           |
| 127 | 0050 8698 |
| 128 |           |
| 129 | 0052 4702 |
| 130 | 0054 0002 |
| 131 |           |
| 132 | 0056 4401 |
| 133 | 0058 0C00 |
| 134 |           |
| 135 | 005A 4521 |
| 136 | 005C 0008 |
| 137 |           |
| 138 | 005E 8821 |
| 139 |           |
| 140 | 0060 0921 |
| 141 |           |
| 142 | 0062 2001 |
| 143 | 0064 5400 |
| 144 |           |
| 145 | 0066 2102 |
| 146 | 0068 1234 |
| 147 |           |
| 148 | 006A 5402 |
| 149 | 006C 0400 |
| 150 |           |
| 151 | 006E AD31 |
| 152 |           |
| 153 | 0070 2C39 |
| 154 |           |
| 155 | 0072 B742 |
| 156 |           |
| 157 | 0074 B610 |
| 158 |           |
| 159 | 0076 B567 |
| 160 |           |
| 161 |           |
| 162 |           |
| 163 |           |

ANDB R10,RL1:8698  
 AND R2,TW0:4702 0002  
 ORB RH1,I3(4):4401 0C00  
 OR R1,I3(R2):4521 0008  
 XORB RH1,RH2:8821  
 XOP R1,OR2:0921  
 LDB RH1,#%1234:2001 3400  
 LD R2,#%1234:2101 1234  
 LDI RR2,%0400:5402 0400  
 FX R1,R3:AD31  
 FXB RL1,WR3:2C93  
 SBC R2,R4:B742  
 SHCB RH0,RH1:B610  
 ADC R1,R6:B567

|     |                           |           |
|-----|---------------------------|-----------|
| 184 | ADCB RH2,RH4 ; B442       |           |
| 185 |                           | 0078 B442 |
| 186 | LDK R6, #2A ; BD6A        |           |
| 187 |                           | 007A BD6A |
| 188 | LDAR R1, I4 ; 3401 007A   |           |
| 189 |                           | 007C 3401 |
| 190 |                           | 007E 0000 |
| 191 |                           |           |
| 192 | LVA R8 ,%0080 ; 7608 0080 |           |
| 193 |                           | 0080 7608 |
| 194 |                           | 0082 0080 |
| 195 |                           |           |
| 196 | RLDB RH1,RH2 ; BE21       |           |
| 197 |                           | 0084 BE21 |
| 198 | I4 : KRDB RH1,RH3 ; BC31  |           |
| 199 |                           | 0086 BC31 |
| 200 | RLB RL1, #2 ; H29A        |           |
| 201 |                           | 0088 B29A |
| 202 | RL R1 ; B31A              |           |
| 203 |                           | 008A B31A |
| 204 | PLCB RH2, #2 ; B222       |           |
| 205 |                           | 008C B222 |
| 206 | RLC R8 ; B380             |           |
| 207 |                           | 008E B380 |
| 208 | RRB RH2 ; B22C            |           |
| 209 |                           | 0090 B22C |
| 210 | RR R7 ,#2 ;B37E           |           |
| 211 |                           | 0092 B37E |
| 212 | RRCB RH2 ;B224            |           |
| 213 |                           | 0094 B224 |
| 214 | RRC R3 ;B334              |           |
| 215 |                           | 0096 B334 |
| 216 | SDA R2,R1 ; B32B 0001     |           |
| 217 |                           | 0098 B32B |
| 218 |                           | 009A 0001 |
| 219 | SDAB RH2,RH1 ; B22B 0001  |           |
| 220 |                           | 009C B22B |
| 221 |                           |           |
| 222 |                           |           |
| 223 |                           |           |
| 224 |                           |           |
| 225 |                           |           |
| 226 |                           |           |
| 227 |                           |           |
| 228 |                           |           |



```

241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283

009E 0001
00A0 B32F
00A2 0004
00A4 B323
00A6 0001
00A8 B2A3
00AA 0001
00AC B347
00AE 0001
00B0 1602
00B2 01011234
00B6 1204
00B8 00010001
00BC 1A00
00BE 11111111
00C2 1804
00C4 02020202
00C8 1408
00CA 0001000A
00CE E810

SDAL RR2,RR4 : B32F 0004
SDL R2,R1 :B323 0001
SDLB RL2,R1 : B2A3 0001
SULL RR4 , R1 :B347 0001
ADDL RR2, #%01011234 : 1602 01011234
SURL RR4 , #%00010001 : 1204 00010001
DIVL R00 , #%11111111 : 1A00 11111111
MULTL R04, #%02020202 : 1804 02020202
LDL RR8, #%0001000A : 1408 0001000A
JR IRUF, $+%12 : E810
END

UPDATE THE FORWARD REFFERENCED LABELS
007E 0086

```

INITIAL DISTRIBUTION LIST

-----

No Copies

1. Defence Technical Information Center

Cameron Station

Alexandria , Virginia 22314

2

2. Library, Code 2142

Naval Postgraduate School

Monterey, California 93940

2

3. Departement Chairman, Code 52

Departement of Computer Science

Naval Postgraduate School

Monterey , California 93940

1

4. Professor Lyle Cox , Code 52

Departement of Computer Science

Naval Postgraduate School

Monterey, California 93940

1

5. Professor Roger R. Schnell, Code 52

Departement of Computer Science  
Naval Postgraduate School  
Monterey, California 93942 1

7. Hellenic Navy General Staff  
Education Departement/3  
Stratopedon Papagou  
Holargos , Athens, Greece 3

6. LCDR J. Moschovinos HN  
Pandoras 1 Amarrousion  
Athens , Greece 4

U197148

DUDLEY KNOX LIBRARY - RESEARCH REPORTS



5 6853 01067854 3

U197148

